
Table of Contents

Introduction	1.1
前言	1.2
第一章 Introduction	1.3
第二章 Kubernetes API Basics	1.4
第三章 Basics of client-go	1.5
第四章 Using Custom Resources	1.6
第五章 Automating Code Generation	1.7
第六章 Solutions for Writing Operators	1.8
第七章 Shipping Controllers and Operators	1.9
第八章 Custom API Servers	1.10
第九章 Advanced Custom Resources	1.11
Appendix A. Resources	1.12
Index	1.13
关于作者	1.14
后记	1.15

Index

A

- access control
 - best practices, [Packaging Best Practices](#)
 - for production-ready deployment, [Production-Ready Deployments](#)
 - read access, [Getting the Permissions Right](#)
 - role-based access control (RBAC), [How the API Server Processes Requests](#), [Status subresource](#), [Getting the Permissions Right](#), [Delegated Authorization](#)
 - write access, [Getting the Permissions Right](#)
- admission
 - chain, [Admission](#), [Registering](#)
 - configuration, [Registering](#)
 - initializers, [Options and Config Pattern and Startup Plumbing](#), [Plumbing resources](#)
 - mutating, [Validation](#), [Admission](#)
 - order, [Admission](#)
 - plug-in, [How the API Server Processes Requests](#), [Admission](#), [Registering](#)
 - register, [Registering](#)
 - validating, [Validation](#), [Admission](#)
- admission webhooks
 - architecture, [Admission Webhook Architecture](#)
 - example, [Admission Requirements in the Restaurant Example](#)
 - implementing, [Implementing an Admission Webhook](#)
 - overview of, [Admission Webhooks](#)
 - registering, [Registering Admission Webhooks](#)
 - using, [Admission Webhook in Action](#)
- aggregated API server (see aggregation)
- aggregation, [Custom API Servers](#), [Deploying Custom API Servers](#), [Certificates and Trust](#), [Custom Resource Versioning](#), [Setting Up the HTTPS Server](#)
- aggregator (see aggregation)
- alpha versions, [API Versions and Compatibility Guarantees](#)
- Ansible, [Other Packaging Options](#)
- API groups, [API Terminology](#)
- API Machinery features, [Versioning and Compatibility](#), [API Machinery in Depth-Scheme](#) (see also [Kubernetes API](#))
- API servers (see custom API servers)
- API Services, [API Services](#)
- auditing, [Monitoring, instrumentation, and auditing](#)
- authentication, [Delegated Authentication and Trust](#)
- authorization, [Delegated Authorization](#)
- automated builds, [Production-Ready Deployments](#), [Automated Builds and Testing](#)

B

- Ballerina, [Other Packaging Options](#)
- bearer tokens, [Delegated Authentication and Trust](#)
- beta versions, [API Versions and Compatibility Guarantees](#)
- Borg, [Optimistic Concurrency](#)
- builder pattern, [Creating and Using a Client](#)

C

- caching
 - cache coherency issues, [Informers and Caching](#)
 - in-memory, [Informers and Caching-Informers and Caching](#)
 - work queues, [Work Queue](#)
- categories, [Short Names and Categories](#)
- certificates and trust, [Certificates and Trust](#)
- Chang, Eric, [Other Approaches](#)
- charts, [Helm](#)
- Chef, [Other Packaging Options](#)
- CLI-client based operator creation, [Other Approaches](#)
- client sets
 - client expansion, [Client Expansion](#)
 - client options, [Client Options](#)
 - creating, [Creating and Using a Client](#)
 - discovery client, [Client Sets](#)
 - listings and deletions, [Listings and Deletions](#)
 - main interface, [Client Sets](#)
 - role of, [Client Sets](#)
 - status subresources, [Status Subresources: UpdateStatus](#)
 - versioned clients and internal clients, [Client Sets](#)
 - watches, [Watches](#)
- client-gen tags, [client-gen Tags](#)
- client-go
 - API Machinery repository, [API Machinery in Depth-Scheme](#)
 - client sets, [Client Sets-Client Options](#)
 - custom resource access, [Dynamic Client](#)
 - downloading, [The Client Library](#)
 - informers and caching, [Informers and Caching-Work Queue](#)
 - Kubernetes objects in Go, [Kubernetes Objects in Go-spec and status](#)

- repositories, [The Repositories-API Versions and Compatibility Guarantees](#)
- vendoring, [Vendoring-Go Modules](#)
- versioning scheme, [The Client Library](#)
- clients
 - controller-runtime client, [controller-runtime Client of Operator SDK and Kubebuilder](#)
 - creating and using, [Creating and Using a Client](#)
 - dynamic clients, [Dynamic Client](#)
 - loopback client, [Plumbing resources](#)
 - typed clients, [Typed Clients-Typed client created via client-gen](#)
- cloud-native applications
 - example of, [A Motivational Example](#)
 - types of apps running on Kubernetes, [What Does Programming Kubernetes Mean?](#)
- cloud-native languages, [Other Packaging Options](#)
- cnat (cloud-native at) example, [A Motivational Example](#)
- code examples, obtaining and using, [Using Code Examples](#)
- code generation
 - benefits of, [Automating Code Generation](#)
 - calling code generators, [Calling the Generators](#)
 - client-gen tags, [client-gen Tags](#)
 - controlling with tags, [Controlling the Generators with Tags](#)
 - deepcopy-gen tags, [deepcopy-gen Tags](#)
 - global tags for, [Global Tags](#)
 - informer-gen and lister-gen, [informer-gen and lister-gen](#)
 - local tags for, [Local Tags](#)
 - runtime.Object and DeepCopyObject, [runtime.Object and DeepCopyObject](#)
- cohabitation, [API Terminology](#), [Use Cases for Custom API Servers](#)
- command line interface (CLI), [Using the API from the Command Line](#)-[Using the API from the Command Line](#)
- comments and questions, [How to Contact Us](#)
- commercially available off-the-shelf (COTS) apps, [What Does Programming Kubernetes Mean?](#)
- compatibility
 - compatibility guarantees, [API Versions and Compatibility Guarantees](#)
 - formally guaranteed support matrix, [Versioning and Compatibility](#)
 - versioning and, [Versioning and Compatibility](#)
- configuration management systems, [Other Packaging Options](#)
- conflict errors, [Optimistic Concurrency](#)
- connection errors, [Client Options](#)

- continuous integration (CI), [Automated Builds and Testing](#)
- controller-runtime client, [controller-runtime Client of Operator SDK and Kubebuilder](#)
- controllers and operators
 - changing cluster objects or the external world, [Changing Cluster Objects or the External World](#)
 - control loop, [The Control Loop](#)
 - custom controller scope, [Packaging Best Practices](#)
 - documenting with inline docs, [Packaging Best Practices](#)
 - edge- versus level-driven triggers, [Edge- Versus Level-Driven Triggers](#)
 - events, [Events](#)
 - footprint and scalability of, [Packaging Best Practices](#)
 - functions of, [Controllers and Operators](#)
 - lifecycle management, [Lifecycle Management](#)
 - operators, [Operators](#)
 - optimistic concurrency, [Optimistic Concurrency](#)
 - packaging, [Lifecycle Management and Packaging-Packaging Best Practices](#)
 - production-ready deployments, [Production-Ready Deployments-Monitoring, instrumentation, and auditing](#)
 - writing custom, [Solutions for Writing Operators-Uptake and Future Directions](#)
- conversion

,

Inner Structure of a Custom API Server

,

Internal Types and Conversion

,

API Installation

- conversion-gen, [Conversions](#)
- ConversionReview, [Conversion Webhook Architecture](#)
- function

,

Conversions

- naming pattern, [Conversions](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336849881448>)

- lossless, [API Terminology](#)

- on-the-fly, [API Versions and Compatibility Guarantees](#)
- webhooks, [Custom Resource Versioning, Conversion Webhook Architecture](#)
- core group, [TypeMeta](#)
- CoreOS, [Operators](#)
- CRUD verbs, [Versioning and Compatibility](#)
- curl, [Using the API from the Command Line](#)
- custom API servers
 - architecture

,

The Architecture: Aggregation

\-

Delegated Authorization

- aggregation, [\[The Architecture: Aggregation\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336853032152\)](#)
- API services, [\[API Services\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm463368529986824\)](#)
- delegated authentication and trust, [\[Delegated Authentication and Trust\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336852141112\)](#)
- delegated authorization, [\[Delegated Authorization\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336852082968\)](#)
- inner structure of, [\[Inner Structure of a Custom API Server\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336852593128\)](#)

- benefits of, [Use Cases for Custom API Servers](#)
- CRD drawbacks, [Use Cases for Custom API Servers](#)
- CustomResourceDefinition, [Using Custom Resources-controller-runtime Client of Operator SDK and Kubebuilder](#)
- deploying

,

Deploying Custom API Servers

\-

Sharing etcd

- certificates and trust, [Certificates and Trust](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336844899352>)
- deployment manifests, [Deployment Manifests](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336845812776>)
- RBAC setup, [Setting Up RBAC](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336845362968>)
- running insecurely, [Running the Custom API Server Insecurely](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336845083592>)
- sharing etcd, [Sharing etcd](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336844740888>)

- example of, [Example: A Pizza Restaurant](#)

- writing

,

Writing Custom API Servers

\-

Plumbing resources

- admission, [Admission](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336847396440>)
- API installation, [API Installation](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336848136040>)
- conversions, [Conversions](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336850093512>)
- defaulting, [Defaulting](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336849639864>)
- existing option structs, [Options and Config Pattern and Startup Plumbing](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336851504824>)
- first start, [The First Start](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336850418104>)
- internal types and conversion, [Internal Types and Conversion](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336850296824>)
- options and config pattern, [Options and Config Pattern and Startup Plumbing](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336851946328>)
- registry and strategy, [Registry and Strategy](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336848631880>)
- roundtrip testing, [Roundtrip Testing](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336849403896>)
- validation, [Validation](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336849106792>)

- writing API types, [Writing the API Types](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336850241176>)

- custom resource definitions (CRDs)
 - accessing, [A Developer's View on Custom Resources](#)
 - accessing with client-go dynamic client, [Dynamic Client](#)
 - accessing with controller-runtime client, [controller-runtime Client of Operator SDK and Kubebuilder](#)
 - accessing with typed clients, [Typed Clients-Typed client created via client-gen](#)
 - admission webhooks, [Admission Webhooks-Admission Webhook in Action](#)
 - availability of, [Using Custom Resources](#)
 - best practices, [Packaging Best Practices](#)
 - defining, [Using Custom Resources](#)
 - discovery information, [Discovery Information](#)
 - limits of, [Use Cases for Custom API Servers](#)
 - printer columns, [Printer Columns](#)
 - role of, [Operators, Using Custom Resources](#)
 - short names and categories, [Short Names and Categories](#)
 - structural schemas, [Structural Schemas and the Future of CustomResourceDefinitions-Default Values](#)
 - subresources, [Subresources-Scale subresource](#)
 - type definitions, [Type Definitions](#)
 - validating custom resources, [Validating Custom Resources](#)
 - versioning, [Custom Resource Versioning-Seeing Conversion in Action](#)
 - writing with code generators, [Automating Code Generation-Summary](#)
- custom resources (CR) (see custom resource definitions (CRDs))

D

- declarative state management, [Declarative State Management](#)
- decoding, [Inner Structure of a Custom API Server](#)
- deep copies, [deepcopy-gen Tags-runtime.Object and DeepCopyObject](#)
- deep copy, [Kubernetes Objects in Go](#)
- deep-copy
 - deep-copy methods, [Golang package structure](#)
 - deepcopy-gen, [Calling the Generators](#)
- DeepCopyObject tag, [runtime.Object and DeepCopyObject](#)
- defaulting

,

Internal Types and Conversion

,

Defaulting

,

API Installation

,

Default Values

- defaulter-gen, [Calling the Generators](#)
- delegated authentication, [Delegated Authentication and Trust](#)
- delegated authorization, [Delegated Authorization](#)
- dep (vendor tool), [dep](#)
- deployment (controllers and operators)
 - access control, [Getting the Permissions Right](#)
 - automated builds and testing, [Automated Builds and Testing](#)
 - custom controller observability, [Custom Controllers and Observability](#)
 - lifecycle management, [Lifecycle Management](#)
 - overview of, [Lifecycle Management and Packaging](#)
 - packaging best practices, [Packaging Best Practices](#)
 - packaging challenges, [Packaging: The Challenge](#)
 - packaging with Helm, [Helm](#)
 - packaging with Kustomize, [Kustomize](#)
 - packaging with other tools, [Other Packaging Options](#)
 - production-ready overview, [Production-Ready Deployments](#)
- deployment (custom API servers)
 - certificates and trust, [Certificates and Trust](#)
 - deployment manifests, [Deployment Manifests](#)
 - RBAC setup, [Setting Up RBAC](#)
 - running insecurely, [Running the Custom API Server Insecurely](#)
 - sharing etcd, [Sharing etcd](#)
- desired state, [Declarative State Management](#)
- discovery

,

Short Names and Categories

,

controller-runtime Client of Operator SDK and Kubebuilder

- endpoint, [The First Start](#)
- RESTMapper, [REST Mapping](#)
- discovery client, [Client Sets](#)
- discovery mechanism, [Discovery Information](#)
- distributed version control, [Technology You Need to Understand](#)
- dynamic clients, [Dynamic Client](#)

E

- edge-driven triggers, [Edge- Versus Level-Driven Triggers](#)
- encoding, [Inner Structure of a Custom API Server](#)
- errors
 - advanced error behavior of informers, [Informers and Caching](#)
 - cache coherency issues, [Informers and Caching](#)
 - conflict errors, [Optimistic Concurrency](#)
 - connection errors, [Client Options](#)
 - coping with trigger errors, [Edge- Versus Level-Driven Triggers](#)
- event handlers, [Informers and Caching](#)
- event producers, [Edge- Versus Level-Driven Triggers](#)
- event sources, [Edge- Versus Level-Driven Triggers](#)
- events
 - overview of, [Events](#)
 - watch events versus event objects, [Events](#)
- extension patterns
 - aggregated API servers, [Custom API Servers-Sharing etcd](#)
 - custom resource definitions (CRDs), [Using Custom Resources-controller-runtime Client of Operator SDK and Kubebuilder](#)
 - overview of, [Extension Patterns](#)
- external version, [Internal Types and Conversion](#)

F

- feature gate, [Options and Config Pattern and Startup Plumbing, Default Values](#)
- field selector, [Listings and Deletions](#)
- Flant's Shell operator, [Other Approaches](#)
- fuzzers, [Roundtrip Testing](#)

G

- general availability (GA), [Custom Resource Versioning](#)
- generator
 - client-gen, [Status Subresources: UpdateStatus, Scheme, Global Tags](#)

- conversion-gen, [Conversions](#)
- deepcopy-gen, [Calling the Generators](#)
- defaulter-gen, [Calling the Generators](#)
- informer-gen, [informer-gen and lister-gen](#)
- lister-gen, [informer-gen and lister-gen](#)
- generic registry, [Generic registry](#)
- Git, [Technology You Need to Understand](#)
- glide (vendor tool), [glide](#)
- global tags, [Global Tags](#)
- Go modules, [Go Modules](#)
- Go programming language, [Technology You Need to Understand](#), [What Does Programming Kubernetes Mean?](#), [Basics of client-go](#) (see also [client-go](#))
- Go types, [Kubernetes API Types](#), [TypeMeta](#)
- Golang package structure, [Golang package structure](#)
- Golang types, [API Terminology](#), [Typed Clients](#)
- graceful shutdowns, [Client Options](#)
- graceful termination, [Use Cases for Custom API Servers](#)
- GroupVersion (GV), [Client Sets](#)
- GroupVersionKind (GVK), [API Terminology](#), [Kubernetes Objects in Go](#), [Kinds](#), [Conversions](#), [Implementation](#)
- GroupVersionResource (GVR), [API Terminology](#), [Resources](#)

H

- handler chain, [Inner Structure of a Custom API Server](#)
- health checks, [Production-Ready Deployments](#)
- Helm, [Helm](#)
- HTTP interface, [The HTTP Interface of the API Server](#), [Kubernetes API Versioning](#), [How the API Server Processes Requests](#)-[How the API Server Processes Requests](#), [The Client Library](#)
- hub version, [Internal Types and Conversion](#)

I

- impersonation, [Inner Structure of a Custom API Server](#)
- in-cluster config, [Creating and Using a Client](#)
- informer-gen, [informer-gen and lister-gen](#)
- informers
 - caching and, [Informers and Caching](#)-[Informers and Caching](#)
 - overview of, [The Control Loop](#)
 - syncing, [Plumbing resources](#)
 - work queues, [Work Queue](#)
- internal version, [Internal Types and Conversion](#)
- IntOrString, [IntOrString](#) and [RawExtensions](#)

K

- kinds
 - categories of, [API Terminology](#)
 - formatting of, [Kinds](#)
 - function of, [API Terminology](#)
 - GroupVersionKind (GVK), [API Terminology](#), [Kinds](#)
 - kinds versus resources, [API Terminology](#)
 - living in multiple API groups, [API Terminology](#)
 - relation to Go type, packages, and group names, [TypeMeta](#)
 - typed clients and, [Anatomy of a type](#)
- klog, [Logging](#)
- ksonnet, [Other Packaging Options](#)
- kube-aggregator, [The Architecture: Aggregation](#), [Delegated Authorization](#)
- kube-apiserver, [Type Definitions](#), [The Architecture: Aggregation](#), [Delegated Authorization](#), [Sharing etcd](#), [Setting Up the HTTPS Server](#)
- kube-controller-manager, [Events](#), [Informers and Caching](#)
- kube-dns, [Using the API from the Command Line](#)
- kube-scheduler, [Events](#)
- kube-system, [Events](#), [Using the API from the Command Line](#)
- Kubebuilder
 - additional resources, [Kubebuilder](#)
 - base directory, [Bootstrapping](#)
 - bootstrapping, [Bootstrapping](#)
 - business logic, [Business Logic-Business Logic](#)
 - commands
 - kubebuilder create api, [Bootstrapping](#)
 - kubebuilder init, [Bootstrapping](#)
 - controller-runtime client of, [controller-runtime Client of Operator SDK and Kubebuilder](#)
 - create api command, [Bootstrapping](#)
 - custom resource definition, [Bootstrapping](#)
 - custom resource installation and validation, [Bootstrapping](#)
 - dedicated namespace creation, [Bootstrapping](#)
 - local operator launch, [Bootstrapping](#)
 - versions, [Kubebuilder](#)
- kubeconfig, [Creating and Using a Client](#)
- kubectl, [Events](#), [The API Server](#), [Using the API from the Command Line](#), [Creating and Using a Client](#), [Discovery Information](#), [Validating Custom Resources](#)
- kubectl api-resources, [Using the API from the Command Line](#), [Short Names and Categories](#)
- kubectl api-versions, [Using the API from the Command Line](#)
- kubectl apply, [Packaging: The Challenge](#), [Kustomize](#)
- kubectl apply -f, [Bootstrapping](#), [Bootstrapping](#), [Bootstrapping](#)
- kubectl create -f, [Running the Custom API Server Insecurely](#)
- kubectl delete, [Running the Custom API Server Insecurely](#)

- [kubectl get --raw, Using the API from the Command Line](#)
- [kubectl get apiservices, Running the Custom API Server Insecurely](#)
- [kubectl get at example-at, Pruning Versus Preserving Unknown Fields](#)
- [kubectl get at,pods, Business Logic](#)
- [kubectl get crds, Bootstrapping, Bootstrapping, Bootstrapping](#)
- [kubectl get ds, Short Names and Categories](#)
- [kubectl get pod, Defaulting](#)
- [kubectl logs, Logging](#)
- [kubectl logs example-at-pod, Business Logic](#)
- [kubectl proxy, Using the API from the Command Line](#)
- [kubectl scale --replicas 3, Scale subresource](#)
- [kubecuddler, Other Approaches](#)
- [kubelet, Events, Optimistic Concurrency, Declarative State Management](#)
- [Kubernetes](#)
 - [additional resources, What Does Programming Kubernetes Mean?, The Control Loop, Events, Edge- Versus Level-Driven Triggers, General](#)
 - [API versioning, Kubernetes API Versioning](#)
 - [controllers and operators, Controllers and Operators-Operators](#)
 - [documentation, A Motivational Example, Versioning and Compatibility](#)
 - [ecosystem for, Ecosystem](#)
 - [extension patterns, Extension Patterns](#)
 - [local development environment, What Does Programming Kubernetes Mean?](#)
 - [meaning of programming Kubernetes, What Does Programming Kubernetes Mean?](#)
 - [native app example, A Motivational Example](#)
 - [optimistic concurrency in, Optimistic Concurrency](#)
 - [prerequisites to learning, Technology You Need to Understand](#)
 - [programming in Go, What Does Programming Kubernetes Mean?](#)
 - [types of apps running on, What Does Programming Kubernetes Mean?](#)
 - [versions discussed, Ecosystem](#)
- [Kubernetes API](#)
 - [API Machinery repository, API Machinery](#)
 - [API versioning, Kubernetes API Versioning, Versioning and Compatibility](#)
 - [architecture and core responsibilities, The API Server](#)
 - [benefits of, What Does Programming Kubernetes Mean?](#)
 - [command line control, Using the API from the Command Line-Using the API from the Command Line](#)
 - [declarative state management, Declarative State Management](#)
 - [Go types repository, Kubernetes API Types](#)
 - [HTTP interface of, The HTTP Interface of the API Server](#)
 - [request processing, How the API Server Processes Requests-How the API Server Processes Requests, Client Options](#)
 - [terminology, API Terminology-API Terminology](#)
- [Kubernetes objects in Go](#)
 - [ObjectMeta, ObjectMeta](#)

- overview of, [Kubernetes Objects in Go](#)
- spec and a status section, [spec and status](#)
- TypeMeta, [TypeMeta](#)
- KUDO, [Other Approaches](#)
- Kustomize, [Kustomize](#)
- kutil, [Other Approaches](#)

L

- label selector, [Listings and Deletions](#)
- leader-follower/standby model, [Production-Ready Deployments](#)
- least-privileges principle, [Getting the Permissions Right](#)
- legacy group, [TypeMeta](#)
- level-driven triggers, [Edge- Versus Level-Driven Triggers](#)
- lifecycle management, [Lifecycle Management](#)
- lister-gen, [informer-gen](#) and [lister-gen](#)
- local development environment, [What Does Programming Kubernetes Mean?](#)
- local tags, [Local Tags](#)
- logging, [Production-Ready Deployments](#), [Logging](#)
- long-running requests, [Client Options](#)

M

- manifest files, [Kustomize](#)
- masters, [Deployment Manifests](#)
- Metacontroller, [Other Approaches](#)
- metadata, [ObjectMeta](#)
- monitoring and logging, [Production-Ready Deployments](#), [Custom Controllers and Observability](#)
- mutating plug-ins, [Admission](#)

O

- ObjectTyper, [Strategy](#)
- ObjectMeta, [ObjectMeta](#)
- OLM (Operator Lifecycle Management), [Lifecycle Management](#)
- Omega (Google research paper), [Optimistic Concurrency](#)
- OpenAPI schema language, [Validating Custom Resources](#)
- Operator SDK
 - additional resources, [Business Logic](#)
 - bootstrapping, [Bootstrapping](#)
 - business logic, [Business Logic](#)

- controller-runtime client of, [controller-runtime Client of Operator SDK and Kubebuilder](#)
- installing, [The Operator SDK](#)
- OperatorHub.io, [Operators](#)
- operators

(

see

also controllers and operators)

- building with Operator SDK, [The Operator SDK-Business Logic](#)
- following sample-controller, [Following sample-controller-Business Logic](#)
- implementing with Kubebuilder, [Kubebuilder-Business Logic](#)
- other approaches to writing, [Other Approaches](#)
- overview of, [Operators](#)
- preparation for writing, [Preparation](#)
- writing custom, [Solutions for Writing Operators](#)
- optimistic concurrency, [Optimistic Concurrency](#)
- option-config pattern, [Options and Config Pattern and Startup Plumbing](#), [Setting Up the HTTPS Server](#)

P

- package management, [Technology You Need to Understand](#), [Kubernetes API Types](#), [dep](#), [Helm](#)
- packaging
 - best practices, [Packaging Best Practices](#)
 - challenges of, [Packaging: The Challenge](#)
 - lifecycle management, [Lifecycle Management](#)
 - other options for, [Other Packaging Options](#)
 - with Helm, [Helm](#)
 - with Kustomize, [Kustomize](#)
- parallel scheduler architecture, [Optimistic Concurrency](#)
- Plumi, [Other Packaging Options](#)
- polling, [Edge- Versus Level-Driven Triggers](#)
- post-start hook, [Options and Config Pattern and Startup Plumbing](#)
- printer columns, [Printer Columns](#)
- priority queues, [Work Queue](#)
- Prometheus, [Monitoring, instrumentation, and auditing](#)
- protocol buffers (protobuf), [Creating and Using a Client](#)
- pruning, [Pruning Versus Preserving Unknown Fields](#)
- Puppet, [Other Packaging Options](#)

Q

- questions and comments, [How to Contact Us](#)

R

- rate limiting, [Client Options](#)
- read access, [Getting the Permissions Right](#)
- reflection, [Scheme](#)
- registry, [Options and Config Pattern and Startup Plumbing](#)
- relist period, [Informers and Caching](#)
- remote procedure calls (RPCs), [Events](#)
- replica integer value, [Scale subresource](#)
- repositories
 - API Machinery, [API Machinery](#)
 - API versions and compatibility guarantees, [API Versions and Compatibility Guarantees](#)
 - client library, [The Client Library](#)
 - creating and using clients, [Creating and Using a Client](#)
 - importing, [The Repositories](#)
 - Kubernetes API Go types, [Kubernetes API Types](#)
 - third-party applications, [Vendoring](#)
 - versioning and compatibility, [Versioning and Compatibility](#)
- request processing, [How the API Server Processes Requests](#)-[How the API Server Processes Requests](#), [Client Options](#)
- resource version, [Optimistic Concurrency](#)
- resource version conflict errors, [Optimistic Concurrency](#)
- resources
 - example Kubernetes API space, [API Terminology](#)
 - formatting of, [Resources](#)
 - GroupVersionResource (GVR), [API Terminology](#), [Resources](#)
 - namespaces versus cluster-scoped, [Resources](#)
 - overview of, [API Terminology](#)
 - resources versus kinds, [API Terminology](#)
 - subresources, [API Terminology](#)
- REST client, [Client Sets](#)
- REST config, [Client Sets](#), [Informers and Caching](#), [Dynamic Client](#)
- REST mapping, [API Terminology](#), [REST Mapping](#)
- REST verbs, [The Client Library](#)
- resync period, [Informers and Caching](#)
- role-based access control (RBAC), [How the API Server Processes Requests](#), [Status subresource](#), [Getting the Permissions Right](#), [Delegated Authorization](#), [Setting Up RBAC](#)
- Rook operator kit, [Other Approaches](#)
- roundtrippable conversion, [Internal Types and Conversion](#), [Roundtrip Testing](#)
- runtime.Object, [Kubernetes Objects in Go](#), [Scheme](#), [runtime.Object and DeepCopyObject](#)

S

- Salt, [Other Packaging Options](#)
- sample-controller
 - bootstrapping, [Bootstrapping](#)
 - business logic implementation, [Business Logic-Business Logic](#)
 - implementing operators following, [Following sample-controller](#)
- scale subresource, [Scale subresource](#)
- schema, structural, [Structural Schemas and the Future of CustomResourceDefinitions](#)
- schemes, [Scheme](#)
- semantic versioning (semver), [Versioning and Compatibility](#), [Go Modules](#)
- server request processing, [How the API Server Processes Requests](#)-[How the API Server Processes Requests](#), [Client Options](#)
- server-side printing, [Printer Columns](#)
- service account, [Deployment Manifests](#)
- shared informer factory, [Informers and Caching](#)
- short names, [Short Names and Categories](#)
- Site Reliability Engineers (SREs), [Operators](#)
- spec and a status section, [spec and status](#), [Status subresource](#)
- specifications (specs), [Declarative State Management](#)
- state change
 - declarative state management, [Declarative State Management](#)
 - detecting, [Edge- Versus Level-Driven Triggers](#)
- status (observed state), [Declarative State Management](#)
- status subresources, [Status Subresources: UpdateStatus](#), [Status subresource](#)
- storage versions, [API Versions and Compatibility Guarantees](#)
- stores, [Informers and Caching](#)
- strategy, [Strategy](#)
- structural schemas
 - controlling pruning, [Controlling Pruning](#)
 - default values, [Default Values](#)
 - IntOrString and RawExtensions, [IntOrString and RawExtensions](#)
 - overview of, [Structural Schemas and the Future of CustomResourceDefinitions](#)
 - pruning versus preserving unknown fields, [Pruning Versus Preserving Unknown Fields](#)
- subject access review, [Delegated Authorization](#), [Generic registry](#)
- subresources, [API Terminology](#), [Subresources](#)-[Scale subresource](#)

T

- testing, [Automated Builds and Testing](#)
- third-party applications, [Vendoring](#)
- throttling

,

Client Options

- burst, [Client Options](#)
- queries per second, [Client Options](#)
- timeouts, [Client Options](#)
- triggers
 - coping with errors, [Edge- Versus Level-Driven Triggers](#)
 - edge- versus level-driven triggers, [Edge- Versus Level-Driven Triggers](#)
- type definitions, [Type Definitions](#)
- type system, [API Machinery in Depth](#)
- typed clients, [Typed Clients](#)-Typed client created via client-gen
- TypeMeta, [TypeMeta](#)

U

- UNIX tooling, for packaging, [Other Packaging Options](#)
- user agents, [Client Options](#)

V

- validating plug-ins, [Admission](#)
- validation, [Validation](#)
- vendoring
 - dep, [dep](#)
 - glide, [glide](#)
 - Go modules, [Go Modules](#)
 - role of, [Vendoring](#)
 - tools for, [Vendoring](#)
- version control, [Technology You Need to Understand](#)
- versioning
 - conversion webhook architecture, [Conversion Webhook Architecture](#)
 - conversion webhook deployment, [Deploying the Conversion Webhook](#)
 - conversion webhook implementation, [Conversion Webhook Implementation](#)
 - example, [Revising the Pizza Restaurant](#)
 - HTTPS server setup, [Setting Up the HTTPS Server](#)
 - overview of, [Custom Resource Versioning](#)
 - process of, [Seeing Conversion in Action](#)
- versions, in Kubernetes API, [API Terminology](#), [Versioning and Compatibility](#)

W

- WATCH verb, [The Client Library](#)
- watches, [Events](#), [Watches](#), [Client Options](#)

- webhooks
 - admission webhooks, [Admission Webhooks](#)-[Admission Webhook in Action](#)
 - conversion webhook architecture, [Conversion Webhook Architecture](#)
 - conversion webhook deployment, [Deploying the Conversion Webhook](#)
 - conversion webhook implementation, [Conversion Webhook Implementation](#)
- work queues, [The Control Loop](#), [Work Queue](#)
- write access, [Getting the Permissions Right](#)

X

- x-kubernetes-embedded-object: true, [IntOrString](#) and [RawExtensions](#)
- x-kubernetes-int-or-string: true, [IntOrString](#) and [RawExtensions](#)
- x-kubernetes-preserve-unknown-fields: true, [Controlling Pruning](#)

Y

- YAML manifests, [Packaging](#): The Challenge
- ytt, [Other Packaging Options](#)

Z

- Zalando's Kopf, [Other Approaches](#)

前言

欢迎来到Kubernetes编程，感谢您选择与我们共度时光。在我们深入研究之前，让我们快速获得一些管理和组织的东西。我们也将分享编写本书的动机。

谁应该读这本书

您是开发人员的云端本地人，或AppOps或命名空间管理员希望从Kubernetes中获得最大收益。香草设置不再适合您了，您可能已经了解了[扩展点](#)。好。你是在正确的地方。

我们为什么写这本书

自2015年初以来，我们两人一直在为Kubernetes做贡献，写作，教学和使用。我们为Kubernetes开发了工具和应用程序，并为Kubernetes开发了几次研讨会。在某些时候我们说，“为什么我们不写一本书？”这将允许更多的人，以他们自己的节奏异步地学习如何编程Kubernetes。我们在这里。我们希望您在阅读本书时能够像编写本书一样有趣。

生态系统

在事情的宏伟计划，Kubernetes生态系统仍处于早期阶段。虽然Kubernetes在2018年初确立了自己作为管理容器（及其生命周期）的行业标准，但仍然需要有关如何编写本机应用程序的良好实践。基础构建块（例如 `client-go`，自定义资源和云原生编程语言）已到位。然而，大部分知识都是部落的，分散在人们的脑海中，分散在成千上万的Slack频道和StackOverflow答案中。

注意

在撰写本文时，Kubernetes 1.15是最新的稳定版本。编译后的示例应该适用于较旧的版本（低至1.12），但我们将代码基于较新版本的库，对应于1.14。一些更高级的CRD功能需要运行1.13或1.14集群，第9章甚至1.15中的CRD转换。如果您无法访问最近的群集，强烈建议您使用本地工作站上的[Minikube](#)或[kind](#)。

您需要了解的技术

这个中级书籍需要对一些开发和系统管理概念的最小理解。在深入研究之前，您可能需要查看以下内容：

- 包管理

该本书中的工具通常具有多个依赖项，您需要通过安装某些软件包来满足这些依赖项。因此，需要了解机器上的包管理系统。这可能是容易在Ubuntu / Debian的系统，百胜在CentOS / RHEL系统或端口或BREW在MacOS。无论是什么，请确保您知道如何安装，升级和删除软件包。

- 去

Git已经确立了自己的标准分布式版本控制。如果您已经熟悉CVS和SVN但尚未使用Git，那么您应该这样做。Jon Loeliger和Matthew McCullough（O'Reilly）使用Git进行版本控制是一个很好的起点。与Git一起，[GitHub网站](#)是您开始使用自己的托管存储库的绝佳资源。要了解GitHub，请查看[他们的培训产品](#)和[相关的交互式教程](#)。

- 走

Kubernetes用[Go](#)写的。在过去的几年中，Go已成为许多初创公司和许多与系统相关的开源项目的首选新编程语言。这本书不是教你Go，但它告诉你如何用Go编程Kubernetes。您可以学习浏览各种不同的资源，从[Go网站](#)上的在线文档到博客文章，演讲和一些书籍。

本书中使用的约定

本书使用以下印刷约定：

- 斜体

表示新术语，URL，电子邮件地址，文件名和文件扩展名。

- `Constant width`

用于程序列表，以及段落内部，用于引用程序元素，如变量或函数名称，数据库，数据类型，环境变量，语句和关键字。也用于命令和命令行输出。

- **Constant width bold**

显示应由用户按字面输入的命令或其他文本。

- *Constant width italic*

显示应使用用户提供的值替换的文本或由上下文确定的值。

小费

此元素表示提示或建议。

注意

该元素表示一般性说明。

警告

此元素表示警告或警告。

使用代码示例

这个本书可以帮助您完成工作。你可以在GitHub的组织在整本书中使用的代码示例[本书](#)。

通常，如果本书提供了示例代码，您可以在程序和文档中使用它。除非您复制了大部分代码，否则您无需与我们联系以获得许可。例如，编写使用本书中几个代码块的程序不需要许可。出售或分发O'Reilly书籍中的示例CD-ROM需要获得许可。通过引用本书并引用示例代码来回答问题不需要许可。

将本书中的大量示例代码合并到产品文档中需要获得许可。

我们感谢，但不要求，归属。归属通常包括标题，作者，出版商和ISBN。例如：“ Michael Hausenblas和Stefan Schimanski (O'Reilly) 的Kubernetes 编程。版权所有2019 Michael Hausenblas和Stefan Schimanski。”

如果您认为您对代码示例的使用超出了合理使用范围或上述许可范围，请随时通过 permissions@oreilly.com与我们联系。

本书中使用的Kubernetes清单，代码示例和其他脚本可通过[GitHub获得](#)。您可以克隆这些存储库，转到相关的章节和配方，并按原样使用代码。

O'Reilly在线学习

注意

近40年来，[O'Reilly Media](#)提供技术和业务培训，知识和洞察力，帮助公司取得成功。

我们独特的专家和创新者网络通过书籍，文章，会议和在线学习平台分享他们的知识和专业知识。O'Reilly的在线学习平台为您提供按需访问实时培训课程，深入学习路径，交互式编码环境以及来自O'Reilly和200多家其他出版商的大量文本和视频。有关更多信息，请访问<http://oreilly.com>。

如何联系我们

请向发布商提出有关本书的评论和问题：

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (在美国或加拿大)
- 707-829-0515 (国际或当地)
- 707-829-0104 (传真)

我们有一本本书的网页，其中列出勘误表，示例和任何其他信息。您可以通过<https://oreil.ly/pr-kubernetes>访问此页面。

请发送电子邮件至bookquestions@oreilly.com发表评论或询问有关本书的技术问题。

有关我们的书籍，课程，会议和新闻的更多信息，请访问我们的网站<http://www.oreilly.com>。

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

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

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

致谢

一个很大的“谢谢你！”向Kubernetes社区致敬，他们开发了这样一款出色的软件，并成为了一大群人 - 开放，善良，随时准备提供帮助。此外，我们非常感谢我们的技术评审员：Ahmed Belgana, Michael Gasch, Dimitris Gkanatsios, Mingding Han, Jess Males, MaxNeunhöffer, Ewout Prangsma和Adrien Trouillaud。您都提供了极具价值和可操作性的反馈，使本书对读者更具可读性和实用性。感谢您的时间和精力！

迈克尔想对他那令人敬畏和支持的家庭表示最深切的谢意：我邪恶的聪明而有趣的妻子Anneliese; 我们的孩子Saphira, Ranya和Iannis; 和我们几乎还是小狗的史努比。

Stefan感谢他的妻子Clelia，当他再次“在书上工作”的时候，他感到非常支持和鼓励。没有她，这本书就不会在这里。如果你在书中发现拼写错误，很可能他们很自豪地由两只猫Nino和Kira贡献。

最后但同样重要的是，两位作者都感谢O'Reilly团队，特别是弗吉尼亚威尔逊，他们在编写本书的过程中引导我们，确保我们按时交付并保证质量。

第1章简介

编程Kubernetes对不同的人来说意味着不同的东西。在本章中，我们将首先确定本书的范围和重点。此外，我们将分享关于我们正在运营的环境以及您需要提供什么样的假设，理想情况下，从本书中获益最多。我们将通过编程Kubernetes，Kubernetes本地应用程序是什么来定义我们的意思，并通过查看具体示例，了解它们的特征。我们将讨论控制器和操作器的基础知识，以及事件驱动的Kubernetes原理如何控制平面功能。准备？我们来吧。

编程Kubernetes意味着什么？

我们假设您可以访问正在运行的Kubernetes集群，例如Amazon EKS，Microsoft AKS，Google GKE或其中一个OpenShift产品。

小费

You将花费大量时间在笔记本电脑或桌面环境中进行本地开发；也就是说，您正在开发的Kubernetes集群是本地的，而不是云或数据中心。在本地开发时，您可以使用多种选项。根据您的操作系统和其他首选项，您可以选择一个（或甚至更多）以下解决方案在本地运行Kubernetes：[kind](#)，[k3d](#)或[Docker Desktop](#)。¹

我们还假设您是Go程序员 - 也就是说，您具有Go编程语言的经验或至少基本熟悉。现在是一个好时机，如果这些假设中的任何一个不适用于你，那么训练：对于Go，我们推荐Alan AA Donovan和Brian W. Kernighan (Addison-Wesley) 的[Go Go Programming Language](#)和Katherine的[Go并发](#) Cox-Buday (奥莱利)。对于 Kubernetes，查看以下一本或多本书：

- 曼宁的[行动总督](#)
- [Kubernetes: Up and Running, 第2版](#)，Kelsey Hightower等。（O'Reilly）的
- 与 John Arundel和Justin Domingus (O'Reilly) 合作的[Kubernetes](#)的[Cloud Native DevOps](#)
- Brendan Burns和Craig Tracey管理[Kubernetes](#) (O'Reilly)
- Sébastien Goasguen和Michael Hausenblas (O'Reilly) 的[Kubernetes Cookbook](#)

注意

为什么我们专注于Go中的Kubernetes编程？好吧，类比可能在这里有用：Unix是用C编程语言编写的，如果你想为Unix编写应用程序或工具，你会默认为C.另外，为了扩展和定制Unix - 即使你是使用C以外的语言 - 您至少需要能够阅读C.

现在，Kubernetes和许多相关的云原生技术，从容器运行时到监控，如Prometheus，都是用Go编写的。我们相信大多数原生应用程序都是基于Go的，因此我们在本书中专注于它。如果您更喜欢其他语言，请关注[kubernetes-client](#) GitHub组织。它可能会继续以您最喜欢的编程语言包含客户端。

通过在本书的上下文中，“编程Kubernetes”我们的意思如下：您将开发一个Kubernetes本机应用程序，它直接与API服务器交互，查询资源状态和/或更新其状态。我们并不意味着运行现成的应用程序，如WordPress或Rocket Chat或您最喜欢的企业CRM系统，通常称为商用现成（COTS）应用程

序。此外，在[第7章中](#)，我们并没有真正关注运营问题，而是主要关注开发和测试阶段。所以，简而言之，这本书是关于发展的真正的云原生应用程序。[图1-1](#)可能会帮助您更好地吸收它。

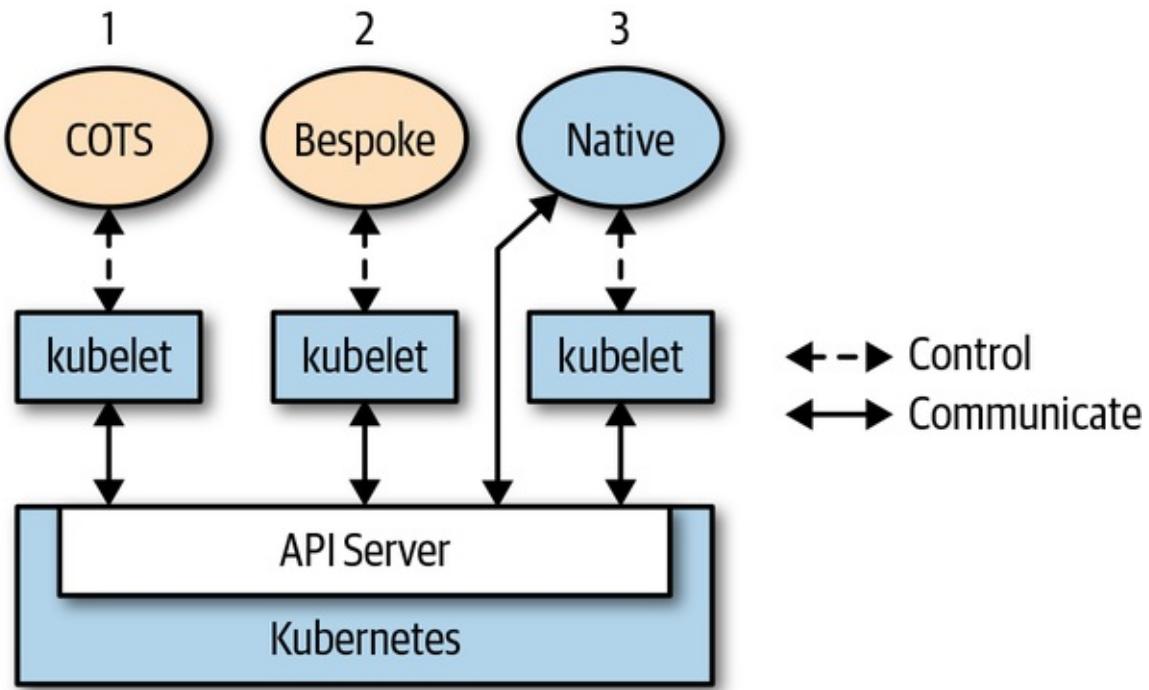


图1-1。在Kubernetes上运行的不同类型的应用程序

如 你可以看到，有你可以使用的不同风格：

- 拿一个像Rocket Chat这样的COTS并在Kubernetes上运行它。应用程序本身并不知道它在Kubernetes上运行，通常不必。Kubernetes控制应用程序的生命周期 - 查找节点以运行，提取图像，启动容器，执行运行状况检查，装载卷等等 - 就是这样。
- 拿一个定制的应用程序，你从头开始编写的东西，无论是否考虑到Kubernetes作为运行时环境，并在Kubernetes上运行它。适用与COTS相同的操作方式。
- 我们在本书中关注的案例是云本机或Kubernetes本机应用程序，它完全了解它在Kubernetes上运行并在某种程度上利用Kubernetes API和资源。

该 您根据Kubernetes API支付的价格得到回报：一方面您获得了可移植性，因为您的应用程序现在可以在任何环境中运行（从内部部署到任何公共云提供商），另一方面您可以从中受益Kubernetes提供的清洁，声明机制。

让我们现在转到一个具体的例子。

一个激励的例子

至演示了Kubernetes原生应用程序的强大功能，让我们假设您要实现 `at` - 也就是说，在给定时间安排执行命令。

我们称这个 `cnat` 或云原生 `at`，它的工作原理如下。假设你想 `echo "Kubernetes native rocks!"` 在2019年7月3日凌晨2点执行命令。这是你要做的事情 `cnat` :

```
$ cat cnat-rocks-example.yaml
apiVersion: cnat.programming-kubernetes.info/v1alpha1
亲切的: 在
元数据:
  名称: cnrex
规格:
  时间表: "2019-07-03T02:00:00Z"
  容器:
    - 名称: shell
      图像: centos: 7
      command:
        - "bin/bash"
        - "-c"
        - echo "Kubernetes native rocks!"

$ kubectl apply -f cnat-rocks-example.yaml
cnat.programming-kubernetes.info/cnrex创建
```

在幕后，涉及以下组件：

- 调用的自定义资源 `cnat.programming-kubernetes.info/cnrex`，表示计划。
- 控制器在正确的时间执行调度的命令。

此外，`kubectl` CLI UX 的插件很有用，允许通过像 `kubectl at "02:00 Jul 3"`
`echo ``"Kubernetes native rocks!"` 这样的命令进行简单处理不会在本书中写这个，但您可以参考[Kubernetes文档获取说明](#)。

在整本书中，我们将使用此示例来讨论Kubernetes的各个方面，其内部工作以及如何扩展它。

对于更高级的例子8和9，我们将模拟集群与比萨比萨饼店和一流的对象。有关详细信息，请参阅[“示例：比萨餐厅”](#)。

扩展模式

Kubernetes是一个功能强大且内在可扩展的系统。通常，有多种方法可以自定义和/或扩展 Kubernetes：使用[配置文件](#)和控制平面组件（如 `kubelet` Kubernetes API服务器）的标志，以及通过许多已定义的扩展点：

- 所谓的[云提供商](#)，传统上是树中的控制器管理器的一部分。从1.11开始，Kubernetes通过提供[与云集成的自定义 cloud-controller-manager 流程](#)，使树外开发成为可能。云提供商允许使用特定于云提供商的工具，如负载平衡器或虚拟机（VM）。
- `kubelet` 用于[网络](#)，[设备](#)（如GPU），[存储](#)和[容器运行时的二进制插件](#)。
- 二进制 `kubectl` 插件。
- API服务器中的访问扩展，例如[带有webhooks的动态准入控制](#)（参见[第9章](#)）。
- 自定义资源（请参阅[第4章](#)）和自定义控制器；请参阅以下部分。
- 自定义API服务器（请参阅[第8章](#)）。
- 调度程序扩展，例如使用[webhook](#)来实现您自己的调度决策。

- 使用webhook进行身份验证。

在本书的上下文中，我们将重点关注自定义资源，控制器， webhook和自定义API服务器，以及 Kubernetes 扩展模式。如果您对其他扩展点感兴趣，例如存储或网络插件，请查看[官方文档](#)。

现在您已经对Kubernetes扩展模式和本书的范围有了基本的了解，让我们转到Kubernetes控制平面的核心，看看我们如何扩展它。

控制器和操作员

在本节中，您将了解Kubernetes中的控制器和操作员以及它们的工作原理。

根据[Kubernetes术语表](#)，控制器实现一个控制循环，通过API服务器观察集群的共享状态，并进行更改以尝试将当前状态移至所需状态。

在我们深入了解控制器的内部工作之前，让我们来定义我们的术语：

- 控制器可以对核心资源（例如部署或服务）执行操作，这些资源通常是控制平面中[Kubernetes控制器管理器](#)的一部分，或者可以监视和操作用户定义的自定义资源。
- 操作员是编码器，它编码一些操作知识，例如应用程序生命周期管理，以及[第4章](#)中定义的自定义资源。

当然，鉴于后者的概念是基于前者，我们首先考虑控制器，然后讨论操作员的更专业的情况。

控制回路

在一般来说，控制循环如下所示：

1. 阅读资源状态，最好是事件驱动（使用手表，如[第3章所述](#)）。有关详细信息，请参阅“[事件](#)”和“[边缘与电平驱动的触发器](#)”。
2. 更改群集或群集外部世界中的对象的状态。例如，启动pod，创建网络端点或查询云API。有关详细信息，请参阅“[更改群集对象或外部世界](#)”。
3. 通过API服务器更新步骤1中资源的状态 etcd。有关详细信息，请参阅“[乐观并发](#)”。
4. 重复循环；回到第1步。

无论您的控制器有多复杂或简单，这三个步骤 - 读取资源状态>更改世界>更新资源状态 - 保持不变。让我们深入探讨一下如何在Kubernetes控制器中实现这些步骤。控制回路[如图1-2所示](#)，其中显示了典型的运动部件，控制器的主回路位于中间。该主循环在控制器进程中持续运行。此过程通常在群集中的pod中运行。

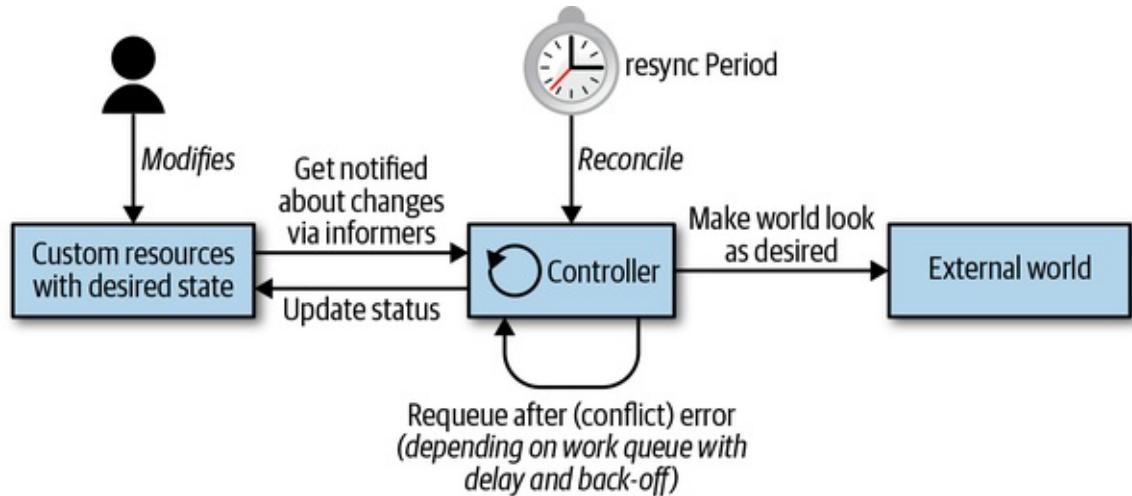


图1-2。Kubernetes控制循环

从架构的角度来看，控制器通常使用以下数据结构（如[第3章](#)详细讨论）：

- 线人

线人以可扩展和可持续的方式观察所需的资源状态。它们还实现了重新同步机制（请参阅[“信息和缓存”](#)以获取详细信息），这些机制强制执行定期协调，通常用于确保群集状态和缓存在内存中的假定状态不会漂移（例如，由于错误或网络问题）。

- 工作队列

基本上，一项工作queue是一个组件，可由事件处理程序用于处理状态更改的排队并帮助实现重试。在 client-go 这个功能是通过所提供的[工作队列包](#)（见[“工作队列”](#)）。在更新世界或写入状态（循环中的步骤2和3）时，如果出现错误，可以重新排队资源，或者仅仅因为我们必须在一段时间后因其他原因重新考虑资源。

对于关于Kubernetes作为声明引擎和状态转换的更正式的讨论，请阅读Andrew Chen和Dominik Tornow 撰写的[“Kubernetes的力学”](#)。

现在让我们仔细看看控制循环，从Kubernetes事件驱动架构开始。

活动

该Kubernetes控制平面大量使用事件和松耦合组件的原理。其他分布式系统使用远程过程调用（RPC）以触发行为。Kubernetes没有。Kubernetes控制器监视API服务器中Kubernetes对象的更改：添加，更新和删除。当发生这样的事件时，控制器执行其业务逻辑。

例如，为了通过部署启动pod，许多控制器和其他控制平面组件一起工作：

1. 部署控制器（内部 `kube-controller-manager`）通知（通过部署通知程序）用户创建部署。它在其业务逻辑中创建副本集。
2. 副本集控制器（再次在内部 `kube-controller-manager`）通知（通过副本集提交者）新副本集并随后运行其业务逻辑，从而创建pod对象。
3. 调度程序（在 `kube-scheduler` 二进制文件内） - 它也是一个控制器 - 通过一个

空 `spec.nodeName` 字段通知pod（通过pod informer）。其业务逻辑将pod放入其调度队列中。

4. 与此同时- `kubelet` 另一个控制器 - 注意到新的pod（通过其pod informer）。但是新pod的 `spec.nodeName` 字段为空，因此与 `kubelet` 节点名称不匹配。它忽略了pod并重新进入睡眠状态（直到下一个事件）。
5. 调度程序将pod从工作队列中取出，并通过更新 `spec.nodeName` pod中的字段并将其写入API服务器，将其调度到具有足够可用资源的节点。
6. 将 `kubelet` 再次由于吊舱更新事件唤醒。它再次将其 `spec.nodeName` 与自己的节点名称进行比较。名称匹配，因此 `kubelet` 启动容器的容器并通过将此信息写入容器状态并返回API服务器来报告容器已启动。
7. 副本集控制器注意到已更改的窗格但无关。
8. 最终pod终止。该 `kubelet` 会注意到这一点，您可以通过API服务器类对象，并设置在吊舱的状态中的“结束”状态，并把它写回API服务器。
9. 副本集控制器注意到已终止的pod并确定必须替换此pod。它删除API服务器上已终止的pod并创建一个新的pod。
10. 等等。

如您所见，许多独立控制循环仅通过API服务器上的对象更改以及这些更改通过触发器触发的事件进行通信。

这些事件从API服务器发送到控制器内的通知器手表（参见“[手表](#)”） - 也就是手表事件的流媒体连接。所有这些对用户来说几乎是不可见的。甚至API服务器审计机制也不会使这些事件可见；只有对象更新可见。但是，当控制器对事件做出反应时，控制器通常会使用日志输出。

观看事件与事件对象

看事件和 `Event` Kubernetes中的顶级对象有两个不同的东西：

- 监视事件通过API服务器和控制器之间的流HTTP连接发送，以驱动线程。
- 顶级 `Event` 对象是类似于pod，部署或服务的资源，具有特殊属性，它具有一小时的生存时间，然后自动清除 `etcd` 。

`Event` 对象仅仅是用户可见的日志记录机制。许多控制器创建这些事件，以便将其业务逻辑的各个方面传达给用户。例如，`kubelet` 报告pod的生命周期事件（即，当容器启动，重新启动和终止时）。

You可以列出自己使用的集群中发生的第二类事件 `kubectl` 。通过发出以下命令，您可以看到 `kube-system` 命名空间中发生了什么：

```
$ kubectl -n kube-system get events
LAST SEEN   FIRST SEEN   COUNT   NAME                                     K
IND
3m          3m           1       kube-controller-manager-master.15932b6fabaa8e5ad   P
od
3m          3m           1       kube-apiserver-master.15932b6fa3f3fbcc      P
od
3m          3m           1       etcd-master.15932b6fa8a9a776        P
od
...
2m          3m           2       weave-net-7nvnf.15932b73e61f5bc6       P
```

od				
2m	3m	2	weave-net-7nvnf.15932b73eefec0b3	P
od				
2m	3m	2	weave-net-7nvnf.15932b73e8f7d318	P
od				

如果如果您想了解有关活动的更多信息，请阅读Michael Gasch的博客文章“[活动，Kubernetes的DNA](#)”，在那里他提供了更多背景和示例。

边缘与水平驱动的触发器

让我们退一步，更加抽象地看看我们如何构建在控制器中实现的业务逻辑，以及为什么Kubernetes选择使用事件（即状态变化）来驱动其逻辑。

有两个原则选项 检测状态变化（事件本身）：

- 边缘驱动的触发器

在发生状态更改的时间点，触发处理程序 - 例如，从无pod到pod运行。

- 水平驱动的触发器

检查状态定期间隔，如果满足某些条件（例如，pod运行），则触发处理程序。

该后者是一种民意调查。它不能很好地适应对象的数量，并且控制器注意到更改的延迟取决于轮询的间隔以及API服务器可以回答的速度。如“[事件](#)”中所述，涉及许多异步控制器，结果是需要很长时间来实现用户期望的系统。

对于许多对象，前一种选择效率更高。延迟主要取决于控制器处理事件中的工作线程数。因此，Kubernetes基于事件（即边缘驱动的触发器）。

在Kubernetes控制平面上，许多组件在API服务器上更改对象，每次更改都会导致事件（即边缘）。我们将这些组件称为事件源或事件生成器。另一方面，在控制器的上下文中，我们对消耗事件感兴趣 - 即何时以及如何对事件做出反应（通过线人）。

在分布式系统中，有许多actor并行运行，并且事件以任何顺序异步进入。当我们有一个错误的控制器逻辑，一些稍微错误的状态机或外部服务失败时，很容易丢失事件，因为我们不完全处理它们。因此，我们必须深入研究如何应对错误。

在图1-3中，您可以看到不同的工作策略：

1. 仅边缘驱动逻辑的示例，其中可能错过第二状态改变。
2. 边缘触发逻辑的一个示例，它在处理事件时始终获得最新状态（即级别）。换句话说，逻辑是边缘触发但是水平驱动。
3. 具有附加重新同步的边缘触发的水平驱动逻辑的示例。

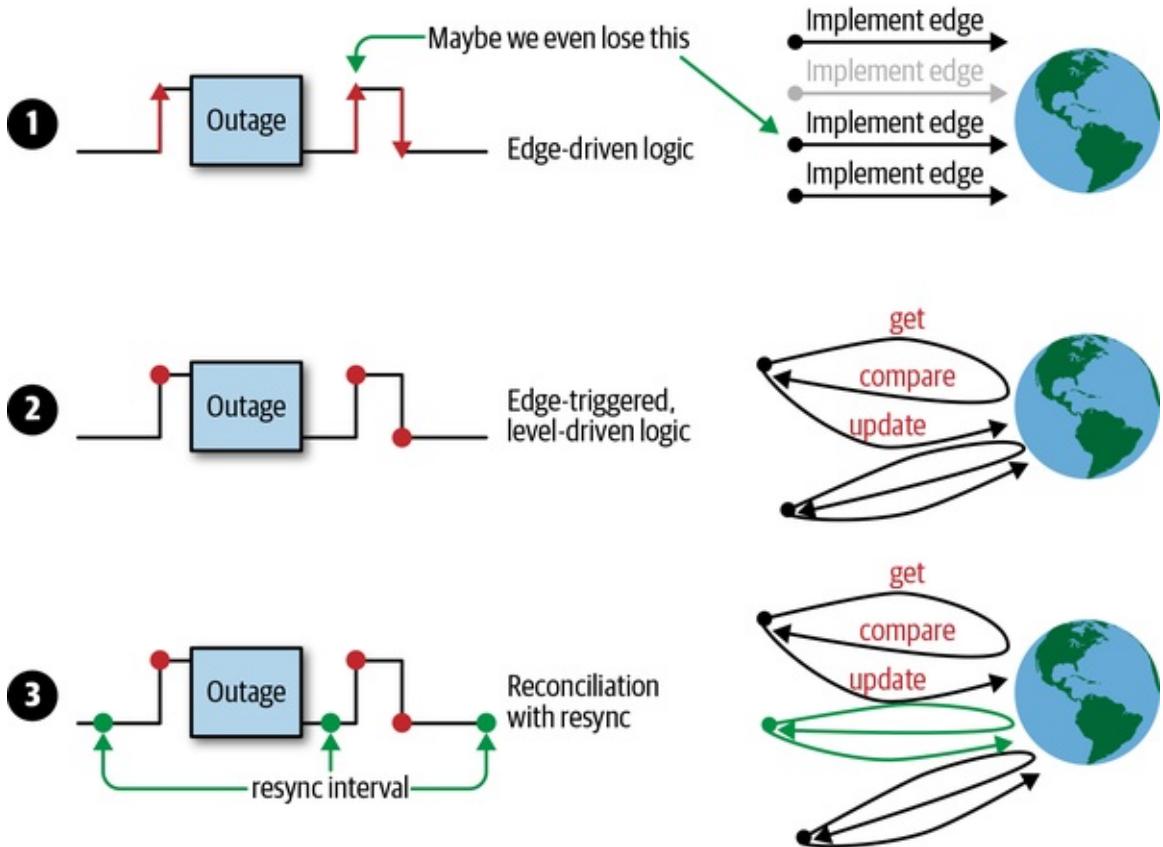


图1-3。触发选项（边缘驱动与电平驱动）

策略1无法很好地应对错过的事件，无论是因为破坏的网络使其丢失事件，还是因为控制器本身存在错误或某些外部云API已关闭。想象一下，副本集控制器只有在终止时才会替换pod。缺少事件意味着副本集将始终以较少的pod运行，因为它永远不会协调整个状态。

策略2在收到另一个事件时从这些问题中恢复，因为它基于集群中的最新状态实现其逻辑。对于副本集控制器，它始终将指定的副本计数与群集中正在运行的pod进行比较。当它丢失事件时，它将在下次收到pod更新时替换所有丢失的pod。

策略3添加连续重新同步（例如，每五分钟）。如果没有pod事件进入，它将至少每五分钟协调一次，即使应用程序运行非常稳定并且不会导致许多pod事件。

鉴于纯边缘驱动触发器的挑战，Kubernetes控制器通常实施第三种策略。

如果你想了解更多关于触发器的起源以及在Kubernetes中使用和解进行关卡触发的动机，请阅读James Bowes的文章[“Kubernetes中的关卡触发和调和”](#)。

最后讨论了检测外部变化并对其作出反应的不同抽象方法。[图1-2控制循环的下一步是更改集群对象或按照规范更改外部世界](#)。我们现在来看看。

更改群集对象或外部世界

在这个阶段，控制器改变它正在监督的对象的状态。例如，[控制器管理](#) ReplicaSet 器中的[控制器](#)正在监督pod。在每个事件（边缘触发）上，它将观察其pod的当前状态，并将其与所需状态（水平驱动）进行比较。

由于更改资源状态的行为是特定于域或任务的，因此我们几乎无法提供指导。相反，我们将继续关注 `ReplicaSet` 我们之前介绍的控制器。`ReplicaSet` 用于部署，相应控制器的底线是：维护用户定义数量的相同pod副本。也就是说，如果播客数量少于用户指定的播放次数（例如，因为播放器已经死亡或者副本值已经增加），控制器将启动新播客。但是，如果有太多的pod，它会选择一些用于终止。控制器的整个业务逻辑经由可用的[replica_set.go包](#)，和与状态改变（编辑为清楚起见）转到代码交易以下摘录：

```
// manageReplicas checks and updates replicas for the given ReplicaSet.
// It does NOT modify <filteredPods>.
// It will requeue the replica set in case of an error while creating/deleting pods

func (rsc *ReplicaSetController) manageReplicas(
    filteredPods []*v1.Pod, rs *apps.ReplicaSet,
) error {
    diff := len(filteredPods) - int(*rs.Spec.Replicas)
    rsKey, err := controller.KeyFunc(rs)
    if err != nil {
        utilruntime.HandleError(
            fmt.Errorf("Couldn't get key for %v %#v: %v", rsc.Kind, rs, err),
        )
        return nil
    }
    if diff < 0 {
        diff *= -1
        if diff > rsc.burstReplicas {
            diff = rsc.burstReplicas
        }
        rsc.expectations.ExpectCreations(rsKey, diff)
        klog.V(2).Infof("Too few replicas for %v %s/%s, need %d, creating %d",
            rsc.Kind, rs.Namespace, rs.Name, *(rs.Spec.Replicas), diff,
        )
        successfulCreations, err := slowStartBatch(
            diff,
            controller.SlowStartInitialBatchSize,
            func() error {
                ref := metav1.NewControllerRef(rs, rsc.GroupVersionKind)
                err := rsc.podControl.CreatePodsWithControllerRef(
                    rs.Namespace, &rs.Spec.Template, rs, ref,
                )
                if err != nil && errors.IsTimeout(err) {
                    return nil
                }
                return err
            },
        )
        if skippedPods := diff - successfulCreations; skippedPods > 0 {
            klog.V(2).Infof("Slow-start failure. Skipping creation of %d pods, " +
                "decrementing expectations for %v %v/%v",
                skippedPods, rsc.Kind, rs.Namespace, rs.Name,
            )
        }
    }
}
```

```

        )
        for i := 0; i < skippedPods; i++ {
            rsc.expectations.CreationObserved(rsKey)
        }
    }
    return err
} else if diff > 0 {
    if diff > rsc.burstReplicas {
        diff = rsc.burstReplicas
    }
    klog.V(2).Infof("Too many replicas for %v %s/%s, need %d, deleting %d",
                    rsc.Kind, rs.Namespace, rs.Name, *(rs.Spec.Replicas), diff,
                    )
}

podsToDelete := getPodsToDelete(filteredPods, diff)
rsc.expectations.ExpectDeletions(rsKey, getPodKeys(podsToDelete))
errCh := make(chan error, diff)
var wg sync.WaitGroup
wg.Add(diff)
for _, pod := range podsToDelete {
    go func(targetPod *v1.Pod) {
        defer wg.Done()
        if err := rsc.podControl.DeletePod(
            rs.Namespace,
            targetPod.Name,
            rs,
        ); err != nil {
            podKey := controller.PodKey(targetPod)
            klog.V(2).Infof("Failed to delete %v, decrementing " +
                "expectations for %v %s/%s",
                podKey, rsc.Kind, rs.Namespace, rs.Name,
            )
            rsc.expectations.DeletionObserved(rsKey, podKey)
            errCh <- err
        }
    }(pod)
}
wg.Wait()

select {
case err := <-errCh:
    if err != nil {
        return err
    }
default:
}
}
return nil
}

```

您可以看到控制器计算行中规范和当前状态之间的差异 `diff := len(filteredPods) - int(rs.Spec.Replicas)`，然后根据具体情况实现两种情况：

- `diff < 0` : 复制品太少; 必须创建更多的pod。
- `diff > 0` : 复制品太多; 必须删除pod。

它还实施了一种策略来选择最不利于删除它们的pod `getPodsToDelete`。

但是，更改资源状态并不一定意味着资源本身必须是Kubernetes集群的一部分。换句话说，控制器可以改变位于Kubernetes之外的资源的状态，例如云存储服务。例如，[AWS Service Operator](#)允许您管理AWS资源。除此之外，它还允许您管理S3存储桶 - 即，S3控制器正在监控存在于Kubernetes之外的资源（S3存储桶），状态更改反映了其生命周期中的具体阶段：创建了一个S3存储桶，在某些时候删除。

这应该说服您使用自定义控制器，您不仅可以管理核心资源（如pod）和自定义资源（如我们的 `cnat` 示例），还可以计算或存储Kubernetes之外的资源。这使控制器具有非常灵活和强大的集成机制，提供了跨平台和环境使用资源的统一方法。

乐观并发

在“控制循环”中，我们在步骤3中讨论了控制器 - 在根据规则更新集群对象和/或外部世界之后将结果规范写入触发控制器在步骤1中运行的资源的状态。

这和其他任何写入（也在步骤2中）都可能出错。在分布式系统中，此控制器可能只是更新资源的众多控制器之一。由于写冲突，并发写入可能会失败。

为了更好地了解正在发生的事情，让我们退一步看看图1-4。2

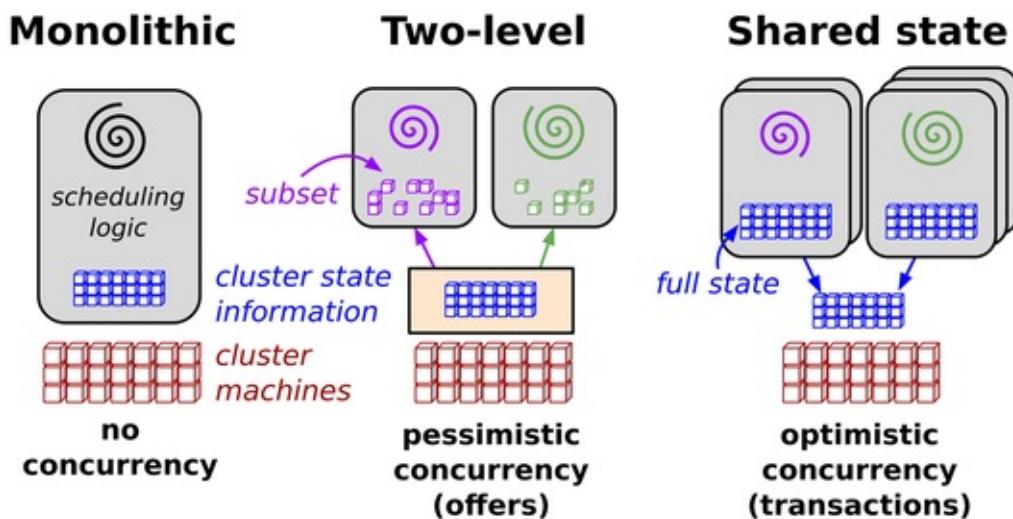


图1-4。调度分布式系统中的体系结构

该 source 定义了Omega的并行调度器架构，如下所示：

我们的解决方案是围绕共享状态构建的新并行调度程序体系结构，使用无锁的乐观并发控制来实现实现可扩展性和性能可伸缩性。这种架构正在谷歌的下一代集群管理系统Omega中使用。

而Kubernetes继承了Borg的许多特性和经验教训，这个特定的事务控制平面功能来自Omega：为了在没有锁的情况下执行并发操作，Kubernetes API服务器使用乐观并发。

简而言之，这意味着如果API服务器检测到并发写入尝试，它将拒绝后两次写入操作。然后由客户端（控制器，调度程序 kubectl 等）来处理冲突并可能重试写操作。

以下演示了Kubernetes中乐观并发的概念：

```

var err error
for retries := 0; retries < 10; retries++ {
    foo, err = client.Get("foo", metav1.GetOptions{})
    if err != nil {
        break
    }

    <更新的环球和 - FOO>

    _, err = client.Update(foo)
    if err != nil && !err.IsConflict {
        continue
    } else if {
        break
    }
}

```

代码显示了一个重试循环，它 `foo` 在每次迭代中获取最新的对象，然后尝试更新世界和 `foo` 状态以匹配 `foo` 规范。在 `update` 通话之前完成的更改是乐观的。

`foo` 来自 `client.Get` 调用的返回对象包含一个资源版本（嵌入式 `ObjectMeta` 结构的一部分- 请参阅[“ObjectMeta”](#)以获取详细信息），它将告诉调用后 `etcd` 的写操作，同时 `client.Update` 集群中的另一个actor编写了该 `foo` 对象。如果是这种情况，我们的重试循环将获得资源版本冲突错误。这意味着乐观并发逻辑失败。换句话说，`client.Update` 电话也是乐观的。

注意

资源版本实际上是 `etcd` 键/值版本。每个对象的资源版本是Kubernetes中包含整数的字符串。这个整数直接来自 `etcd`。`etcd` 维护一个计数器，每次修改一个键（保存对象的序列化）的值时，该计数器都会增加。

在整个API机器代码中，资源版本（或多或少因此）像任意字符串一样处理，但在其上有一些排序。存储整数的事实只是当前 `etcd` 存储后端的实现细节。

让我们看一个具体的例子。想象一下，您的客户端不是群集中修改pod的唯一角色。那里是另一个演员，即 `kubelet` 不断修改某些字段，因为容器不断崩溃。现在你的控制器读取pod对象的最新状态，如下所示：

```

kind: Pod
metadata:
  name: foo

```

```

resourceVersion: 57
spec:
...
status:
...

```

现在假设控制器需要几秒钟来更新世界。七秒钟后，它尝试更新它读取的pod - 例如，它设置了一个注释。同时，`kubelet` 已注意到另一个容器重启并更新了pod的状态以反映出来；也就是说，`resourceVersion` 已经增加到58。

控制器在更新请求中发送的对象具有 `resourceVersion: 57`。API服务器尝试 `etcd` 使用该值设置 pod 的密钥。`etcd` 注意资源版本不匹配，并报告与58冲突58.更新失败。

此示例的底线是，对于您的控制器，您负责实施重试策略并适应乐观操作失败。您永远不知道还有谁可能在操纵状态，无论是其他自定义控制器还是核心控制器（如部署控制器）。

其实质是：冲突错误在控制器中完全正常。总是期待他们并优雅地处理它们。

重要的是要指出乐观并发非常适合基于级别的逻辑，因为通过使用基于级别的逻辑，您可以重新运行控制循环（请参阅[“Edge-Versus Level-Driven Triggers”](#)）。该循环的另一次运行将自动撤消先前失败的乐观尝试的乐观变化，并且它将尝试将世界更新为最新状态。

让我们继续讨论自定义控制器的特定情况（以及自定义资源）：运营商。

运营商

运营商作为Kubernetes中的一个概念，CoreOS于2016年推出。在他的开创性博客文章[“介绍运营商：将操作知识融入软件”](#)中，CoreOS首席技术官Brandon Philips将运营商定义如下：

一个现场可靠性工程师（SRE）是一个通过编写软件来操作应用程序的人。他们是工程师，开发人员，知道如何专门为特定应用领域开发软件。由此产生的软件具有编程到其中的应用程序的操作领域知识。

[...]

我们将这类新的软件称为操作员。Operator是一个特定于应用程序的控制器，它扩展了Kubernetes API，以代表Kubernetes用户创建，配置和管理复杂有状态应用程序的实例。它建立在基本的Kubernetes资源和控制器概念的基础上，但包括领域或特定于应用程序的知识，以自动执行常见任务。

在本书的上下文中，我们将使用飞利浦描述的运算符，更正式地说，要求满足以下三个条件（参见图1-5）：

- 您希望自动化一些特定于领域的操作知识。
- 这种操作知识的最佳实践是已知的并且可以明确 - 例如，在Cassandra操作员的情况下，何时以及如何重新平衡节点，或者在服务网格的操作员的情况下，如何创建一条路线。
- 在运营商的背景下运送的工作件是：
 - 一个一组自定义资源定义（CRD）捕获CRD之后的特定于域的模式和自定义资源，在实例级别上，CRD表示感兴趣的域。

- 自定义控制器，监控自定义资源，可能还有核心资源。例如，自定义控制器可能会启动一个 pod。

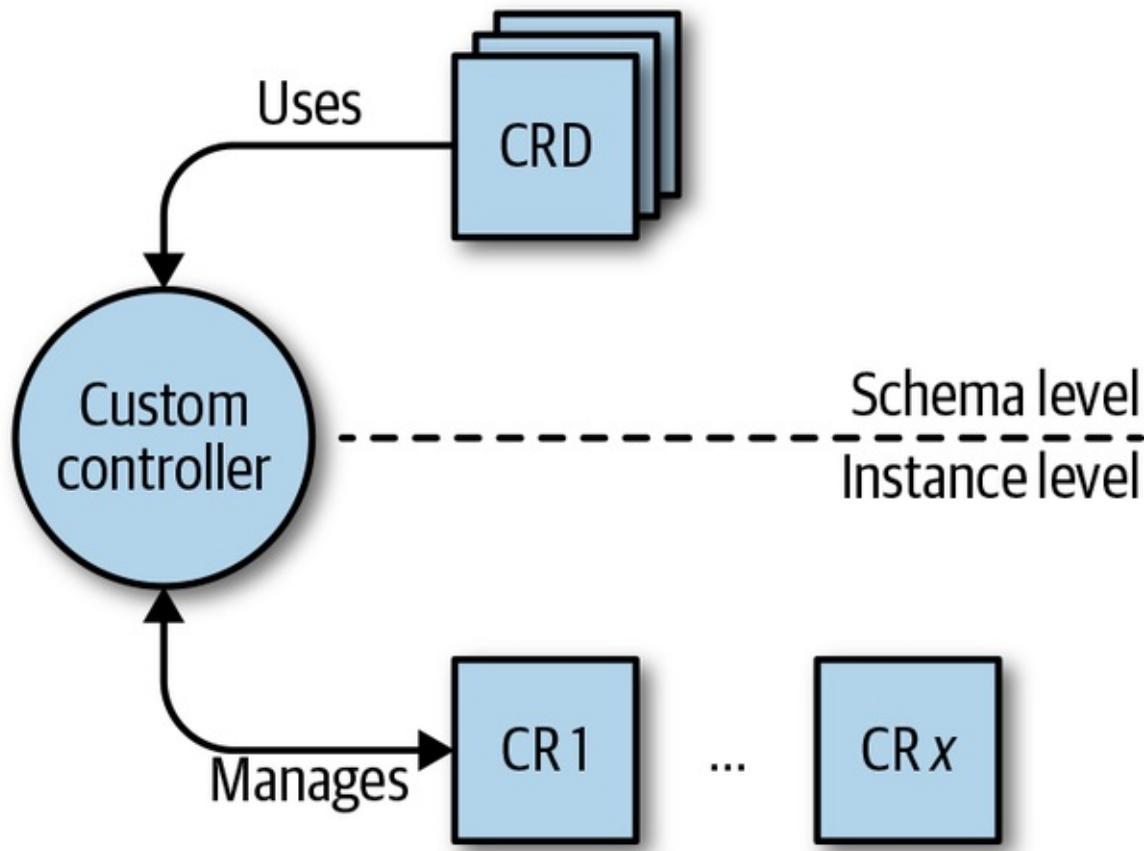


图1-5。运营商的概念

运营商从2016年的概念性工作和原型设计到Red Hat（在2018年收购CoreOS并继续构建该想法）的[OperatorHub.io](#)的推出已经走了很长一段路。在[图1](#)中可以看到[图1-6](#)的截图。2019年中期的中心，有大约17名运营商，随时可以使用。

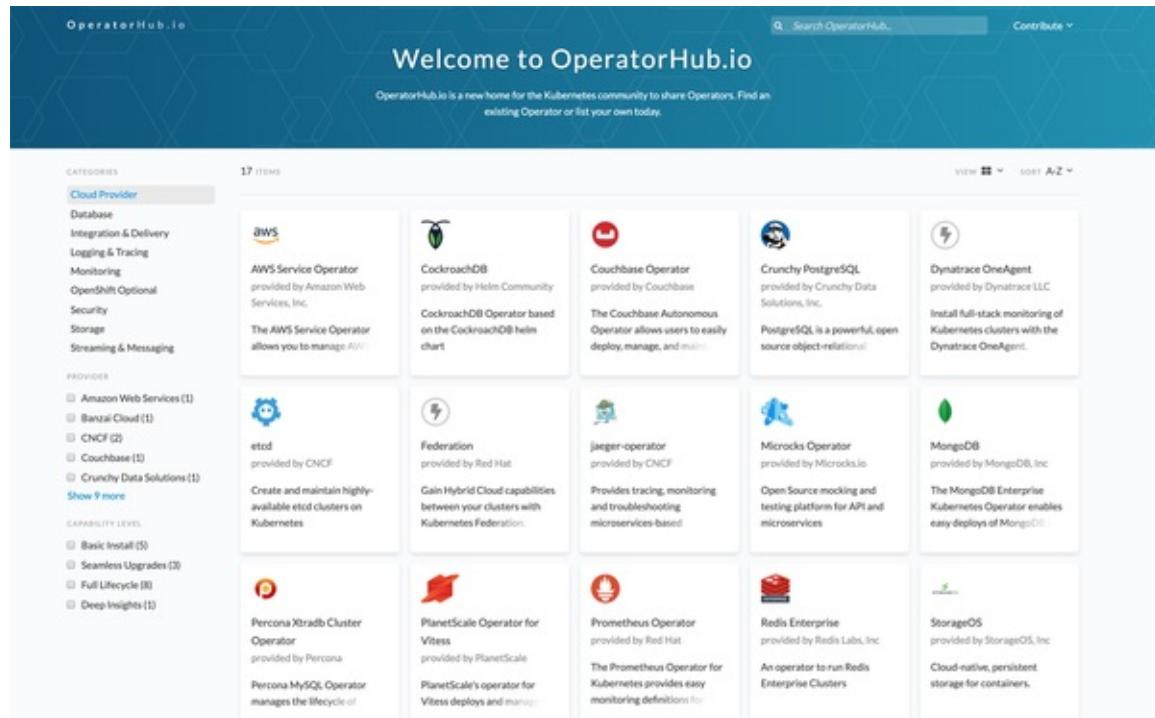


图1-6。OperatorHub.io截图

摘要

在第一章中，我们定义了本书的范围以及我们对您的期望。我们通过编程Kubernetes解释了我们的意思，并在本书的上下文中定义了Kubernetes原生应用程序。作为后续示例的准备，我们还提供了对控制器和操作员的高级介绍。

所以，既然你已经知道了本书的内容以及如何从中获益，那么让我们深入探讨。在下一章中，我们将详细介绍Kubernetes API，API服务器的内部工作方式，以及如何使用命令行工具与API进行交互 curl¹。

¹有关此主题的更多信息，请参阅Megan O'Keefe的“适用于MacOS的Kubernetes开发人员工作流程”，Medium，2019年1月24日；和Alex Ellis的博客文章“Be KinD to yourself”，2018年12月14日。

²来源：“Omega：适用于大型计算集群的灵活，可扩展的调度程序”，作者：Malte Schwarzkopf等人，Google AI，2013。

第2章 Kubernetes API基础知识

在本章中，我们将向您介绍Kubernetes API的基础知识。这包括深入了解API服务器的内部工作，API本身以及如何从命令行与API进行交互。我们将向您介绍Kubernetes API概念，例如资源和种类，以及分组和版本控制。

API服务器

Kubernetes由一组具有不同角色的节点（集群中的机器）组成，[如图2-1所示](#)：主节点上的控制平面由API服务器，控制器管理器和调度程序组成。API服务器是中央管理实体，也是与分布式存储组件直接对话的唯一组件 etcd。

API服务器具有以下核心职责：

- 至服务于Kubernetes API。此API由主组件，工作节点和Kubernetes本机应用程序在内部集群使用，也可由客户端外部使用 `kubectl`。
- 代理群集组件（例如Kubernetes仪表板），或流式传输日志，服务端口或服务 `kubectl exec` 会话。

提供API意味着：

- 读取状态：获取单个对象，列出它们以及流式更改
- 操作状态：创建，更新和删除对象

国家坚持通过 `etcd`。

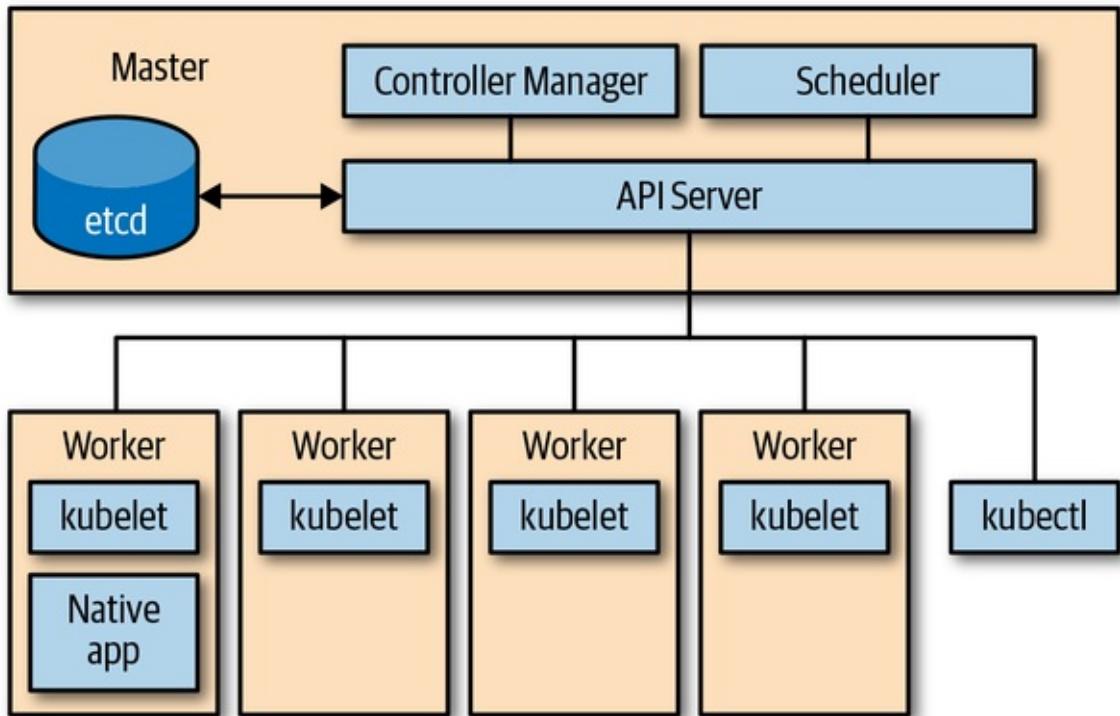


图2-1。Kubernetes建筑概述

Kubernetes的核心是它的API服务器。但是API服务器如何工作？我们首先将API服务器视为黑盒子并仔细查看其HTTP接口，然后我们将继续讨论API服务器的内部工作原理。

API服务器的HTTP接口

从客户的角度来看，API服务器公开了一个RESTful HTTP API，其中包含JSON或[协议缓冲区](#)（简称protobuf）有效负载，主要用于集群内部通信，出于性能原因。

API服务器HTTP接口使用以下[HTTP谓词](#)（或HTTP方法）处理HTTP请求以查询和操作Kubernetes资源：

- **HTTP GET** 谓词用于检索具有特定资源（例如某个窗格）或资源集合或列表（例如，命名空间中的所有窗格）的数据。
- **HTTP POST** 谓词用于创建资源，例如服务或部署。
- **HTTP PUT** 谓词用于更新现有资源 - 例如，更改窗格的容器图像。
- **HTTP PATCH** 谓词用于现有资源的部分更新。阅读Kubernetes文档中的“[使用JSON合并补丁更新部署](#)”，以了解有关可用策略和含义的更多信息。
- **HTTP DELETE** 动词用于以不可恢复的方式销毁资源。

如果你看一下Kubernetes [1.14 API参考](#)，你可以看到不同的HTTP动词。例如，要使用等效的CLI命令列出当前命名空间中的pod `kubectl -n *THENAMESPACE* get pods`，您将发出（参见图2-2）。`GET /api/v1/namespaces/*THENAMESPACE*/pods`

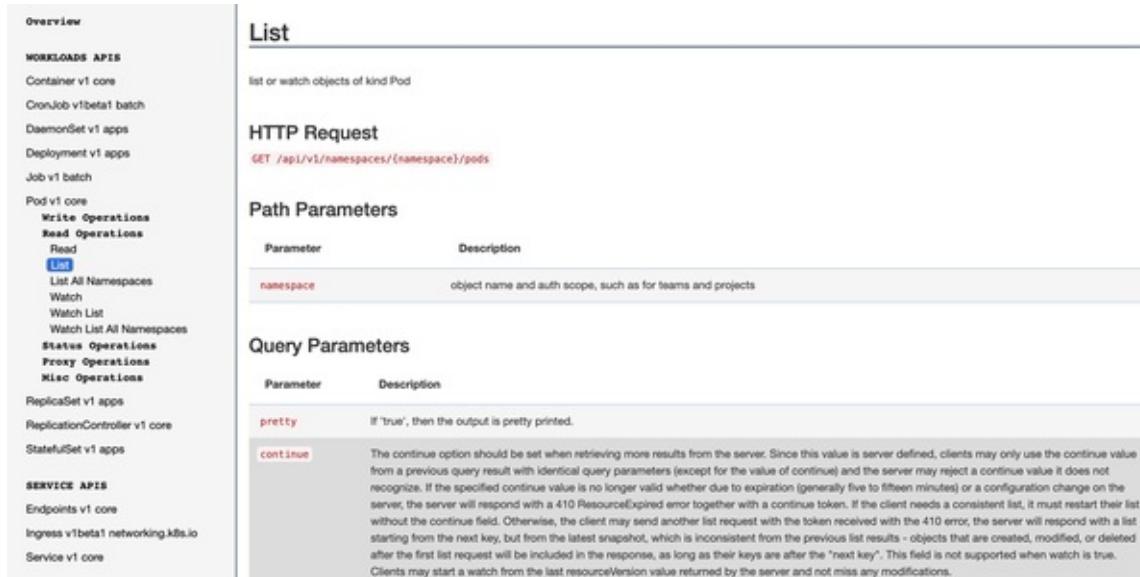


图2-2。运行中的API服务器HTTP接口：列出给定命名空间中的pod

有关如何从Go程序调用API服务器HTTP接口的介绍，请参阅[“客户端库”](#)。

API术语

之前我们进入API业务，让我们首先定义Kubernetes API服务器上下文中使用的术语：

- 类

该实体的类型。每个对象都有一个字段 `Kind` (`kind` JSON中的小写, `Kind` 在Golang中大写), 它告诉客户端, 例如 `kubectl` 它代表一个pod。那里有三类: 对象表示系统中的持久实体 - 例如, `Pod` 或 `Endpoints`。对象具有名称, 其中许多都存在于名称空间中。列表是一种或多种实体的集合。列表具有有限的公共元数据集。例子包括 `PodList`s或 `NodeList`s。当你做一个`kubectl get pods`, 这正是你得到的。专用类用于对象和非持久实体 (如 `/binding` 或) 的特定操作 `/scale`。对于发现, Kubernetes使用 `APIGroup` 和 `APIResource`; 对于错误结果, 它使用 `Status`。

在Kubernetes程序, 直接与Golang类型对应。因此, 作为Golang类型, 种类是单数的, 并以大写字母开头。

- API组

一个 `Kind` 与逻辑相关的s的集合。例如, 所有批次的对象像 `Job` 或 `ScheduledJob` 是批处理 API组中使用。

- 版

每API组可以存在多个版本, 其中大多数都可以。例如, 一个小组首先出现, `v1alpha1` 然后晋升为 `v1beta1` 最终毕业 `v1`。`v1beta1` 可以在每个支持的版本中检索在一个版本 (例如) 中创建的对象。API服务器无损转换为返回请求版本中的对象。从集群用户的角度来看, 版本只是相同对象的不同表示。

小费

没有“`v1` 集群中有一个对象, 集群中有另一个对象 `v1beta1`。”而是, 每个对象都可以作为 `v1` 表示或 `v1beta1` 表示返回, 就像集群用户所希望的那样。

- 资源

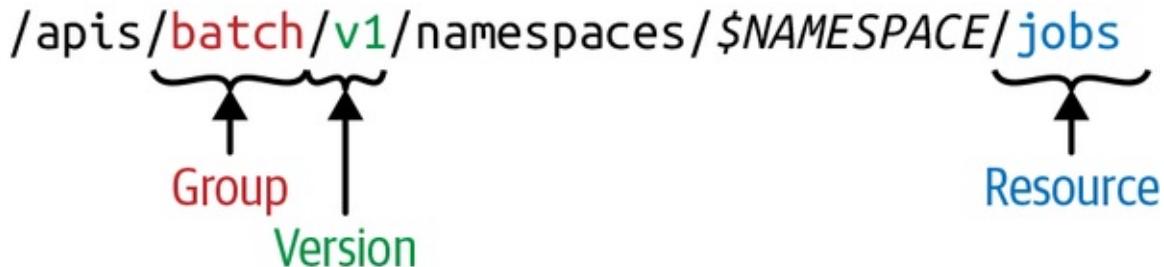
一个通常是小写的, 复数的单词 (例如, `pods`) 标识一组HTTP端点 (路径), 其暴露系统中某个对象类型的CRUD (创建, 读取, 更新, 删除) 语义。常见路径是: 根目录, 例如.../`pods`, 列出该类型的所有实例各个命名资源的路径, 例如.../`pods/nginx`通常, 这些端点中的每一个都返回并接收一种类型 (`PodList` 在第一种情况下为a `Pod`, 在第二种情况下为a)。但在其他情况下 (例如, 在出现错误的情况下), `Status` 会返回一个类型的对象。除了具有完整CRUD语义的主资源之外, 资源还可以有其他端点来执行特定操作 (例如, .../`pod/nginx/port-forward`, .../`pod/nginx/exec`, 或.../`pod/nginx/logs`)。我们调用这些子资源 (参见“[子资源](#)”)。这些通常实现自定义协议而不是REST - 例如, 通过WebSockets或命令式API进行某种流式连接。

小费

资源和种类经常混在一起。请注意明确的区别:

- 资源对应于HTTP路径。
- 种类是这些端点返回和接收的对象类型, 以及持久化的对象类型 `etcd`。

资源始终是API组和版本的一部分，统称为*GroupVersionResource*（或GVR）。GVR唯一地定义HTTP路径。例如，`default` 命名空间中的具体路径是`/apis/batch/v1/namespaces/default/jobs`。[图2-3](#)显示了命名空间资源的示例GVR，a `Job`。



[图2-3。Kubernetes API - GroupVersionResource \(GVR\)](#)

与`jobs` GVR示例相反，群集范围的资源（如节点或命名空间）本身在路径中没有`$ NAMESPACE`部分。例如，`nodes` GVR示例可能如下所示：`/api/v1/nodes`。请注意，名称空间显示在其他资源的HTTP路径中，但也是资源本身，可在`/api/v1/namespaces`访问。

同样对于GVR，每种类型都存在于API组中，具有版本，并通过*GroupVersionKind* (GVK) 进行识别。

同居 - 生活在多个API组中的种类

种同名的，不仅可以在不同的版本中共存，也可以在不同的API组中共存。例如，`Deployment` 在扩展组中以alpha类型开始，最终在其自己的组中升级为稳定版本`apps.k8s.io`。我们称之为同居。虽然在Kubernetes中不常见，但有一些：

- `Ingress`，`NetworkPolicy` 在 `extensions` 和 `networking.k8s.io`
- `Deployment`，`DaemonSet`，`ReplicaSet` 在 `extensions` 和 `apps`
- `Event` 在核心小组和 `events.k8s.io`

GVK和GVR是相关的。GVK在GVR标识的HTTP路径下提供。调用将GVK映射到GVR的过程REST映射。我们将 RESTMappers 在[“REST Mapping”](#)中看到在Golang中实现REST映射。

从全局的角度来看，API资源空间在逻辑上形成了一个具有顶级节点的树，包括`/api`，`/apis`和一些非等级端点，例如`/healthz`或`/metrics`。此API空间的示例呈现[如图2-4所示](#)。请注意，确切的形状和路径取决于Kubernetes版本，多年来趋于稳定。

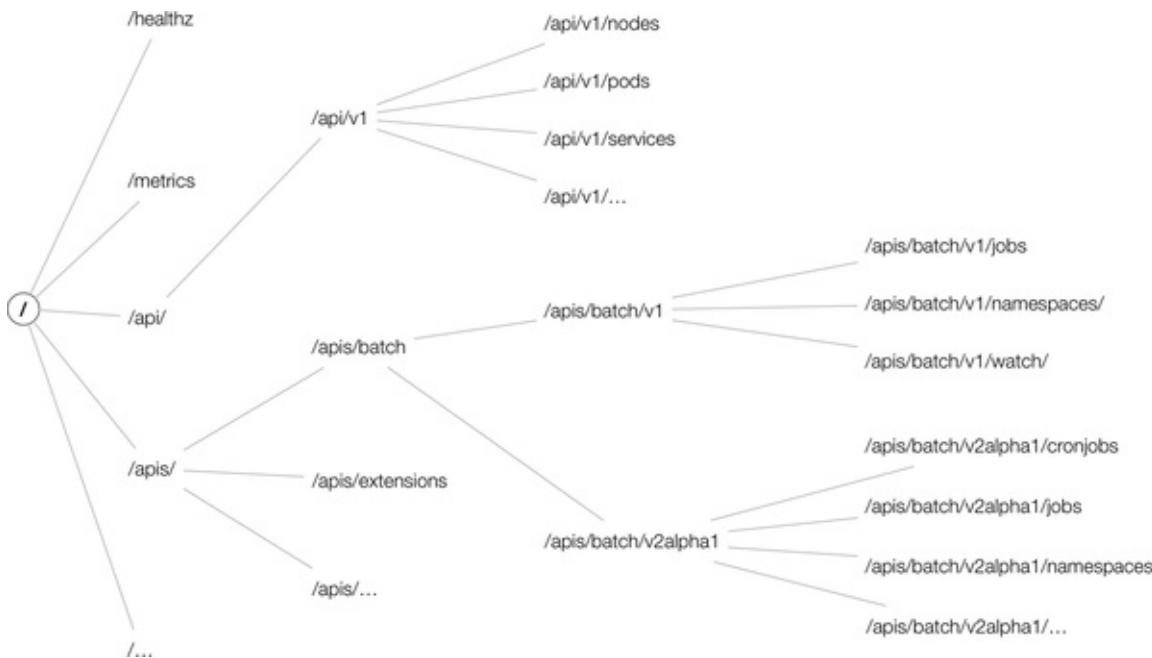


图2-4。一个示例Kubernetes API空间

Kubernetes API版本控制

对于可扩展性原因，Kubernetes支持不同API路径下的多个API版本，例如`/api/v1`或`/apis/extensions/v1beta1`。不同的API版本意味着不同级别的稳定性和支持：

- `v1alpha1` 通常在默认情况下禁用`Alpha`级别（例如）；可以随时删除对功能的支持，恕不另行通知，并且只能在短期测试群集中使用。
- `v2beta3` 默认情况下启用`Beta`级别（例如），这意味着代码已经过充分测试；但是，对象的语义可能会在随后的`beta`或稳定版本中以不兼容的方式发生变化。
- `v1` 对于许多后续版本，稳定（通常可用或`GA`）级别（例如）将出现在已发布的软件中。

让我们来看看如何构造HTTP API空间：在顶层我们区分核心组 - 即`/api/v1`下面的所有内容- 以及`/apis/$NAME/$VERSION`形式的路径中的命名组`*VERSION*`。

注意

由于历史原因，核心小组位于`/apis/core/v1`下，`/api/v1`而不是人们所期望的。在引入API组的概念之前，核心组已存在。

API服务器公开了第三种类型的HTTP路径 - 非资源对齐的路径：群集范围的实体，例如`/metrics`, `/logs`或`/healthz`。此外，API服务器支持手表；也就是说，您可以添加`?watch=true`特定请求，并将API服务器更改为`监视模式`，而不是按设定的时间间隔轮询资源。

声明性国家管理

最API对象区分了所需资源状态的规范和当前时间对象的状态。甲规范，或规范的简称，是一种资源的期望状态的完整描述，并且通常坚持稳定存储，通常`etcd`。

注意

为什么我们说“通常 etcd”？好吧，有Kubernetes发行版和产品，如k3s或微软的AKS，已经取代或正在努力替换 etcd 其他东西。由于Kubernetes控制平面的模块化架构，这很好用。

让我们在API服务器的上下文中更多地讨论规范（期望状态）与状态（观察状态）。

规范描述了您所需的资源状态，您需要通过命令行工具提供，例如 `kubectl` 通过Go代码以编程方式提供。状态描述资源的观察或实际状态，并由控制平面管理，可由核心组件（如控制器管理器）或您自己的自定义控制器管理（请参阅[“控制器和操作员”](#)）。例如，在部署中，您可以指定您希望始终运行20个应用程序副本。部署控制器是控制平面中控制器管理器的一部分，它读取您提供的部署规范并创建一个副本集，然后负责管理副本：它创建相应数量的pod，最终（通过 `kubelet`）导致容器在工作节点上启动。如果任何副本失败，部署控制器将使您知道状态。这就是我们所谓的声明式状态管理 - 也就是说，声明所需的状态并让Kubernetes处理剩下的事情。

当我们开始从命令行开始探索API时，我们将在下一节中看到声明式状态管理。

使用命令行中的API

在本节我们将使用 `kubectl` 并 `curl` 演示Kubernetes API的使用。如果您不熟悉这些CLI工具，现在是安装它们并试用它们的好时机。

首先，让我们看一下资源的期望和观察状态。我们将 `kube-dns` 在 `kube-system` 命名空间中使用可能在每个集群中可用的控制平面组件，CoreDNS插件（旧的Kubernetes版本正在使用）（此输出经过大量编辑以突出显示重要部分）：

```
$ kubectl -n kube-system get deploy / coredns -o =yaml
apiVersion: apps / v1
kind: 部署
元数据:
  名称: coredns
  命名空间: kube-system
  ...
  规格:
    模板:
      规格:
        容器:
          - 名称: coredns
            图片: 602401143452.dkr.ecr.us-east-2.amazonaws.com/eks/coredns:v1.2.2
            ...
  状态:
    复制品: 2
    条件:
      - type: 可用
        status: "True"
        lastUpdateTime: "2019-04-01T16:42:10Z"
        ...

```

正如您可以从此 `kubectl` 命令中看到的那样，在 `spec` 部署的部分中，您将定义一些特征，例如要使用的容器映像以及要并行运行的副本数，并在本 `status` 节中学习了多少个副本。当前时间点实际上正在运行。

为了执行与CLI相关的操作，在本章的其余部分中，我们将使用批处理操作作为运行示例。让我们首先在终端中执行以下命令：

```
$ kubectl proxy --port =8080
开始服务于127.0.0.1:8080
```

此命令将Kubernetes API代理到本地计算机，并且还负责身份验证和授权位。它允许我们通过HTTP直接发出请求并接收JSON有效负载。让我们通过启动我们查询的第二个终端会话来做到这一点 `v1`：

```
$ curl http://127.0.0.1:8080/apis/batch/v1
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "batch/v1",
  "resources": [
    {
      "name": "jobs",
      "singularName": "",
      "namespaced": true,
      "kind": "Job",
      "verbs": [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "update",
        "watch"
      ],
      "categories": [
        "all"
      ]
    },
    {
      "name": "jobs/status",
      "singularName": "",
      "namespaced": true,
      "kind": "Job",
      "verbs": [
        "get",
        "patch",
        "update"
      ]
    }
}
```

```
    ]  
}
```

小费

You不必 curl 与 kubectl proxy 命令一起使用即可获得对Kubernetes API的直接HTTP API访问。

您可以改为使用 kubectl get --raw 命令：例如，替换 curl

```
http://127.0.0.1:8080/apis/batch/v1 为 kubectl get --raw /apis/batch/v1 。
```

将其与 v1beta1 版本进行比较，注意到在查看<http://127.0.0.1:8080/apis/batch> 时，您可以获得批处理API组的受支持版本列表 v1beta1：

```
$ 卷曲http://127.0.0.1:8080/apis/batch/v1beta1  
{  
  "kind": "APIResourceList",  
  "apiVersion": "v1",  
  "groupVersion": "batch/v1beta1",  
  "resources": [  
    {  
      "name": "cronjobs",  
      "singularName": "",  
      "namespaced": true,  
      "kind": "CronJob",  
      "verbs": [  
        "create",  
        "delete",  
        "deletecollection",  
        "get",  
        "list",  
        "patch",  
        "update",  
        "watch"  
      ],  
      "shortNames": [  
        "cj"  
      ],  
      "categories": [  
        "all"  
      ]  
    },  
    {  
      "name": "cronjobs/status",  
      "singularName": "",  
      "namespaced": true,  
      "kind": "CronJob",  
      "verbs": [  
        "get",  
        "patch",  
        "update"  
      ]  
    }  
  ]
```

```

        }
    ]
}

```

如您所见，该 v1beta1 版本还包含 cronjobs 具有该类型的资源 cronJob。在撰写本文时，cron 工作尚未晋升 v1。

如果您想了解群集中支持哪些 API 资源，包括它们的类型，是否为命名空间，以及它们的短名称（主要用于 kubectl 命令行），您可以使用以下命令：

```
$ kubectl api-resources
NAME SHORTNAMES APIGROUP NAMESPACED KIND
绑定 componentstatuses cs true 绑定
configmaps cm true ConfigMap
端点ep true 端点
事件ev true 事件
limitranges 限制 true LimitRange
名称空间ns false 命名空间
节点没有 false 节点
persistentvolumeclaims pvc true PersistentVolumeClaim
persistentvolumes pv false PersistentVolume
pods po true pod
子 true 模板PodTemplate
replicationcontrollers rc true ReplicationController
resourcequotas 配额 true ResourceQuota
秘密 true 的秘密
serviceaccounts sa true ServiceAccount
服务svc true 服务
controllerrevisions 应用程序 true ControllerRevision
daemonsets ds apps true DaemonSet
部署 部署应用程序 true 部署
...

```

以下是一个相关命令，对于确定群集中支持的不同资源版本非常有用：

```
$ kubectl api-versions
admissionregistration.k8s.io/v1beta1
apiextensions.k8s.io/v1beta1
apiregistration.k8s.io/v1
apiregistration.k8s.io/v1beta1
appmesh.k8s.aws/v1alpha1
appmesh.k8s.aws/v1beta1
应用程序/ V1
应用程序/ v1beta1
应用程序/ v1beta2
authentication.k8s.io/v1
authentication.k8s.io/v1beta1
authorization.k8s.io/v1
```

```

authorization.k8s.io/v1beta1
自动缩放/ V1
自动缩放/ v2beta1
自动缩放/ v2beta2
批/ V1
批/ v1beta1
certificates.k8s.io/v1beta1
coordination.k8s.io/v1beta1
crd.amazonaws.com/v1alpha1
events.k8s.io/v1beta1
扩展/ v1beta1
networking.k8s.io/v1
政策/ v1beta1
rbac.authorization.k8s.io/v1
rbac.authorization.k8s.io/v1beta1
scheduling.k8s.io/v1beta1
storage.k8s.io/v1
storage.k8s.io/v1beta1
v1

```

API服务器如何处理请求

现在如果您了解面向外部的HTTP接口，那么我们将重点关注API服务器的内部工作原理。图2-5显示了API服务器中请求处理的高级概述。

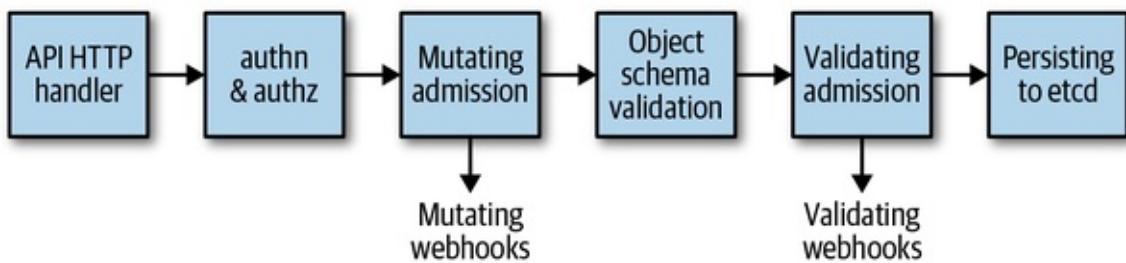


图2-5。Kubernetes API服务器请求处理概述

所以呢实际上当HTTP请求到达Kubernetes API时会发生什么？在较高的层面上，发生以下互动：

1. HTTP请求由注册的过滤器链处理 `DefaultBuildHandlerChain()`。该链在 [k8s.io/apiserver/pkg/server/config.go](#) 中定义，并在稍后详细讨论。它对所述请求应用一系列过滤操作。任一个过滤器通过并附着各信息添加到上下文准确地来说 `ctx.RequestInfo`，与 `ctx` 作为上下文在 Go（例如，通过认证的用户）-或，如果请求不通过过滤器，它返回一个适当的HTTP响应代码说明原因（例如，用户身份验证失败时的 [401 响应](#)）。
2. 接下来，根据HTTP路径，[k8s.io/apiserver/pkg/server/handler.go](#) 中的多路复用器将HTTP请求路由到相应的处理程序。
3. 为每个API组注册了一个处理程序 - 有关详细信息，请参阅[k8s.io/apiserver/pkg/endpoints/groupversion.go](#) 和 [k8s.io/apiserver/pkg/endpoints/installer.go](#)。它接受HTTP请求以及上下文（例如，用户和访问权限）并检索并从 `etcd` 存储中传递请求的对象。

现在让我们仔细看看 `server/config.go` 中设置的过滤器链，以及每个过滤器 `DefaultBuildHandlerChain()` 中发生的事情：

```
func DefaultBuildHandlerChain(apiHandler http.Handler, c *Config) http.Handler {
    h := WithAuthorization(apiHandler, c.Authorization.Authorizer, c.Serializer)
    h = WithMaxInFlightLimit(h, c.MaxRequestsInFlight,
        c.MaxMutatingRequestsInFlight, c.LongRunningFunc)
    h = WithImpersonation(h, c.Authorization.Authorizer, c.Serializer)
    h = WithAudit(h, c.AuditBackend, c.AuditPolicyChecker, LongRunningFunc)
    ...
    h = WithAuthentication(h, c.Authentication.Authenticator, failed, ...)
    h = WithCORS(h, c.CorsAllowedOriginList, nil, nil, nil, "true")
    h = WithTimeoutForNonLongRunningRequests(h, LongRunningFunc, RequestTimeout)
    h = WithWaitGroup(h, c.LongRunningFunc, c.HandlerChainWaitGroup)
    h = WithRequestInfo(h, c.RequestInfoResolver)
    h = WithPanicRecovery(h)
    return h
}
```

所有包都在 [k8s.io/apiserver/pkg](#) 中。更具体地审查：

- `WithPanicRecovery()`

处理恢复和记录恐慌。在 [server/filters/wrap.go](#) 中定义。

- `WithRequestInfo()`

附上一个 `RequestInfo` 上下文。在 [端点/过滤器/requestinfo.go](#) 中定义。

- `WithWaitGroup()`

将所有非长时间运行的请求添加到等待组；用于优雅关机。在 [server/filters/waitgroup.go](#) 中定义。

- `WithTimeoutForNonLongRunningRequests()`

超时非长期运行的请求（最喜欢的 `GET`，`PUT`，`POST`，和 `DELETE` 请求），而相比之下，长期运行的请求，如手表和代理请求。在 [server/filters/timeout.go](#) 中定义。

- `WithCORS()`

提供 [CORS](#) 实现。CORS 是跨源资源共享的缩写，是一种允许嵌入 HTML 页面的 JavaScript 将 XMLHttpRequests 发送到与 JavaScript 发起的域不同的域的机制。在 [server/filters/cors.go](#) 中定义。

- `WithAuthentication()`

尝试将给定请求作为人员或机器用户进行身份验证，并将用户信息存储在提供的上下文中。成功时，`Authorization` 将从请求中删除 HTTP 标头。如果身份验证失败，则返回 HTTP `401` 状态代码。在 [端点/过滤器/authentication.go](#) 中定义。

- `WithAudit()`

使用所有传入请求的审核日志记录信息来装饰处理程序。审计日志条目包含诸如请求的源IP，用户调用操作和请求的命名空间之类的信息。在[录取/审计中](#)定义。

- `WithImpersonation()`

通过检查尝试更改用户的请求（类似于 `sudo`）来处理用户模拟。在[端点/过滤器/impersonation.go中](#)定义。

- `WithMaxInFlightLimit()`

限制飞行中的请求数量。在[server/filters/maxinflight.go中](#)定义。

- `WithAuthorization()`

通过调用授权模块检查权限，并将所有授权请求传递给多路复用器，多路复用器将请求分派给正确的处理程序。如果用户没有足够的权限，则返回HTTP `403` 状态代码。Kubernetes现在使用基于角色的访问控制（RBAC）。在[端点/过滤器/authorization.go中](#)定义。

传递此通用处理程序链后（[图2-5中的第一个框](#)），实际的请求处理开始（即执行请求处理程序的语义）：

- 直接处理对`/`, `/version`, `/apis`, `/healthz`和其他nonRESTful API的请求。
- RESTful资源的请求进入请求管道，包括：

- 入场

传入的对象通过准入链。这个链条有20种不同录取插件。[1](#)每个插件都可以是变异阶段的一部分（参见[图2-5中的第三个框](#)），验证阶段的一部分（参见图中的第四个框），或两者兼而有之。在变异阶段，可以改变传入的请求有效载荷；例如，图像拉策略被设置为 `Always` , `IfNotPresent` , 或 `Never` 取决于入场配置。第二个入场阶段纯粹是为了验证；例如，验证pod中的安全设置，或者在创建该命名空间中的对象之前验证是否存在命名空间。

- 验证

根据大型验证逻辑检查传入对象，该逻辑对系统中的每个对象类型都存在。例如，检查字符串格式以验证服务名称中仅使用有效的DNS兼容字符，或者pod中的所有容器名称是唯一的。

- `etcd` - 支持CRUD逻辑

这里实现了我们在[“API服务器的HTTP接口”](#)中看到的不同动词；例如，更新逻辑从中读取对象 `etcd`，检查没有其他用户在[“乐观并发”](#)意义上修改了对象，如果没有，则将请求对象写入 `etcd`。

我们将在以下章节中更详细地研究所有这些步骤；对于例如：

- 自定义资源

验证在[“验证自定义资源”](#)，在录取[“招生网络挂接”](#)中，和一般的CRUD语义第4章

- Golang原生资源

“验证”中的验证，“录取”中的录取以及“注册表和策略”中CRUD语义的实现

摘要

在本章中，我们首先将Kubernetes API服务器作为黑盒子进行了讨论，并查看了其HTTP接口。然后你学习了如何在命令行上与那个黑盒子进行交互，最后我们打开了黑盒子并探索了它的内部工作原理。到目前为止，您应该知道API服务器如何在内部工作，以及如何使用CLI工具 `kubectl` 与资源进行交互以进行资源探索和操作。

现在是时候将手动交互留在我们身后的命令行，并开始使用Go: `meet client-go` (Kubernetes“标准库”的核心) 进行编程API服务器访问。

1在一个Kubernetes 1.14簇，这些是（以该顺序）：

和。 NamespaceLifecycle `` NamespaceExists `` SecurityContextDeny `` LimitPodHardAntiAffinityTopology `` PodPreset `` LimitRanger `` ServiceAccount `` NodeRestriction `` TaintNodesByCondition `` AlwaysPullImages `` ImagePolicyWebhook `` PodSecurityPolicy `` PodNodeSelector `` Priority `` DefaultTolerationSeconds `` PodTolerationRestriction `` DenyEscalatingExec `` DenyExecOnPrivileged `` EventRateLimit `` ExtendedResourceToleration `` PersistentVolumeLabel `` DefaultStorageClass `` StorageObjectInUseProtection `` OwnerReferencesPermissionEnforcement `` PersistentVolumeClaimResize `` MutatingAdmissionWebhook `` ValidatingAdmissionWebhook `` ResourceQuota `` AlwaysDeny

第3章客户端的基础知识

我们现在将重点关注Kubernetes编程接口在Go。您将学习如何访问众所周知的本机类型（如pod，服务和部署）的Kubernetes API。在后面的章节中，这些技术将扩展到用户定义的类型。但是，我们首先关注每个Kubernetes集群附带的所有API对象。

存储库

该Kubernetes项目在GitHub上的`kubernetes`组织下提供了许多第三方可消费Git存储库。您需要使用域名`k8s.io / ...`（不是`github.com/kubernetes/...`）将所有这些导入到您的项目中。我们将在以下部分介绍这些存储库中最重要的存储库。

客户端库

该Go中的Kubernetes编程接口主要由`k8s.io/client-go`库组成（为简洁起见，我们将其称之为`client-go`前进）。`client-go`是一个典型的Web服务客户端库，支持所有正式属于Kubernetes的API类型。它可以用来执行通常的REST动词：

- 创建
- 得到
- 名单
- 更新
- 删除
- 补丁

这些REST动词中的每一个都使用“[API服务器的HTTP接口](#)”实现。此外，`watch` 支持动词，这对于类似Kubernetes的API是特殊的，并且是与其他API相比的主要区别之一。

`client-go` 是可在GitHub上获得（参见图3-1），并在Go代码中使用`k8s.io/client-go`软件包名称。它与Kubernetes本身并行运送；也就是说，对于每个Kubernetes `1.x.y` 版本，都有一个`client-go` 带有匹配标记的版本`kubernetes-1.x.y`。

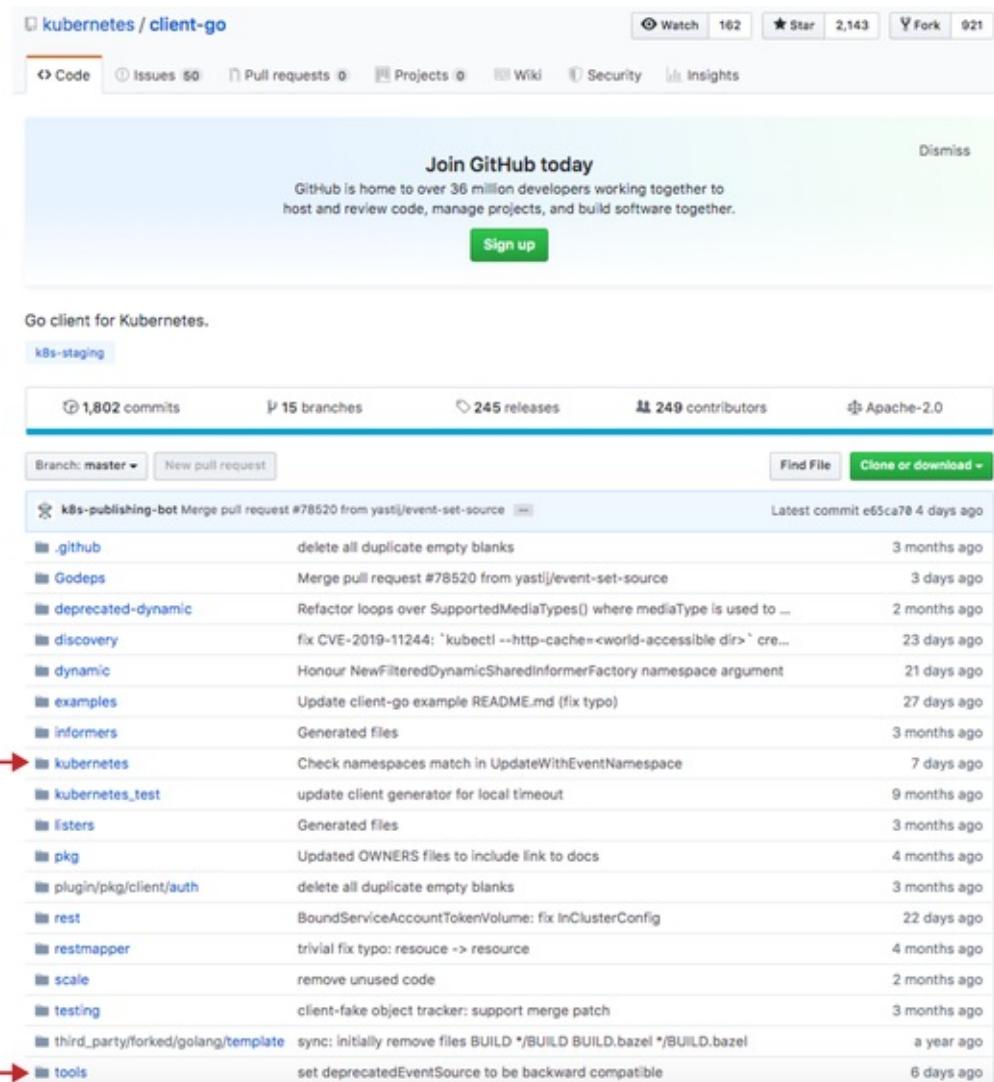


图3-1。GitHub上的client-go存储库

另外，还有语义版本计划。例如，`client-go 9.0.0`匹配Kubernetes 1.12版本，`client-go 10.0.0`匹配Kubernetes 1.13，依此类推。未来可能会有更细粒度的版本。除了Kubernetes API对象的客户端代码外，`client-go`还包含许多通用库代码。这也用于第4章中的用户定义的API对象。有关软件包列表，请参见图3-1。

虽然所有软件包都可以使用，但是大多数与Kubernetes API相关的代码都将使用`tools / clientcmd /`从`kubeconfig`文件中设置客户端，而`kubernetes /`则用于实际的Kubernetes API客户端。我们很快就会看到代码这样做。在此之前，让我们快速浏览一下其他相关的存储库和软件包。

Kubernetes API类型

如我们已经看到，`client-go`拥有客户端接口。`pod`、`服务`和`部署`等对象的Kubernetes API Go类型位于其自己的存储库中。它`k8s.io/api`在Go代码中被访问。

Pod是遗留API组（通常也称为“核心”组）版本的一部分 v1。因此，Pod Go类型位于 `k8s.io/api/core/v1` 中，类似于Kubernetes中的所有其他API类型。看到 图3-2 显示了包列表，其中大多数都与Kubernetes API组及其版本相对应。

实际的Go类型包含在 `types.go` 文件中（例如，`k8s.io/api/core/v1/type.go`）。此外，还有其他文件，其中大部分是由代码生成器自动生成的。

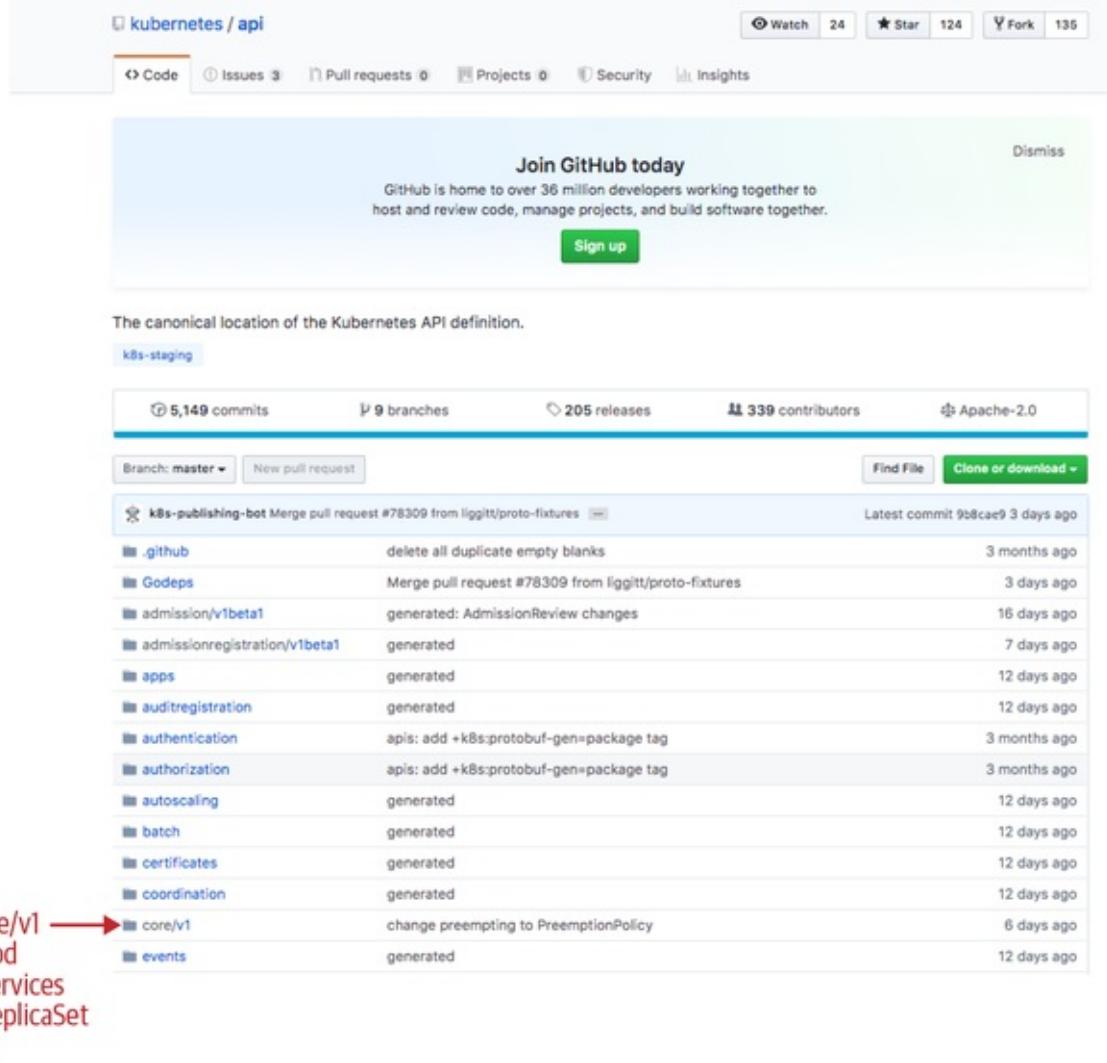


图3-2。GitHub上的API存储库

API机械

持续但并非最不重要的是，还有一个名为 [API Machinery](#) 的第三个存储库，用于 `k8s.io/apimachinery` Go。它包括实现Kubernetes类API的所有通用构建块。API Machinery不仅限于容器管理，因此，例如，它可用于为在线商店或任何其他特定于业务的域构建API。

不过，您将在Kubernetes-native Go代码中遇到很多API Machinery软件包。一个重要的如 `ObjectMeta`，`TypeMeta`，`GetOptions`，和 `ListOptions`（参见图3-3）。

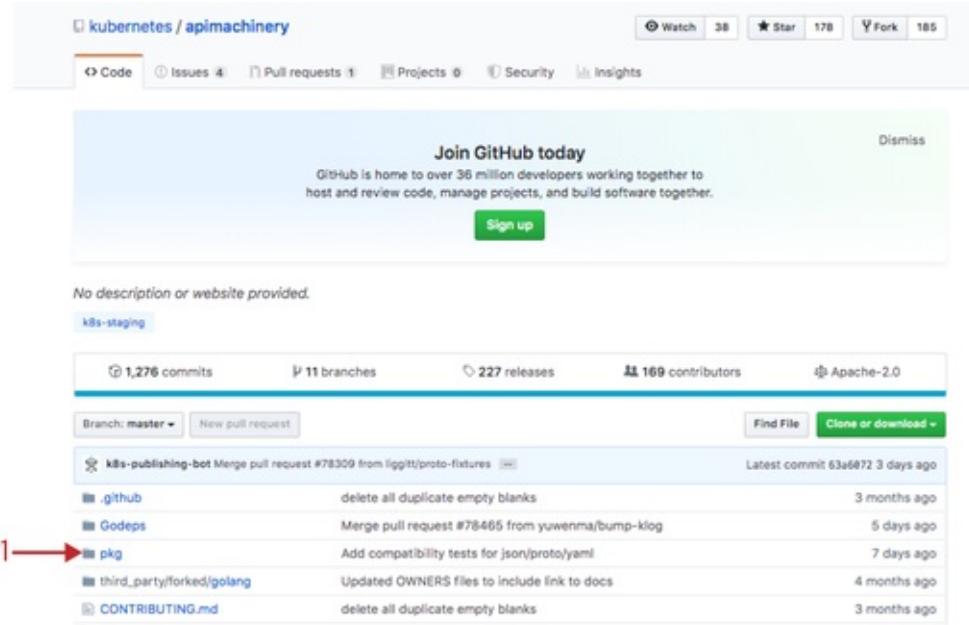


图3-3。GitHub上的API Machinery存储库

创建和使用客户端

现在我们知道创建Kubernetes客户端对象的所有构建块，这意味着我们可以访问Kubernetes集群中的资源。假设您可以访问本地环境中的集群（即，`kubectl`已正确设置并配置了凭据），以下代码说明了如何 `client-go` 在Go项目中使用：

```
import (
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/tools/clientcmd"
    "k8s.io/client-go/kubernetes"
)

kubeconfig = flag.String("kubeconfig", "~/.kube/config", "kubeconfig file")
flag.Parse()
config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
clientset, err := kubernetes.NewForConfig(config)

pod, err := clientset.CoreV1().Pods("book").Get("example", metav1.GetOptions{})
```

代码导入 `meta/v1` 包以获取访问权限 `metav1.GetOptions`。此外，它 `clientcmd` 从中导入 `client-go` 以便读取和解析 `kubeconfig`（即具有服务器名称，凭据等的客户端配置）。然后，它 `client-go``kubernetes` 使用Kubernetes资源的客户端集导入包。

kubeconfig文件的默认位置位于用户主目录的`.kube/config`中。这也是`kubectl`获取Kubernetes集群凭据的地方。

然后使用读取和解析`kubeconfig clientcmd.BuildConfigFromFlags`。我们在整个代码中省略了强制性错误处理，但是`err`如果`kubeconfig`格式不正确，变量通常会包含语法错误。由于语法错误在Go代码中很常见，因此应该正确检查这样的错误，如下所示：

```
config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
if err != nil {
    fmt.Printf("The kubeconfig cannot be loaded: %v\n", err)
    os.Exit(1)
}
```

从`clientcmd.BuildConfigFromFlags`我们得到一个`rest.Config`，您可以在k8s.io/client-go/rest包中找到。这个传递`kubernetes.NewForConfig`给以创建实际的Kubernetes客户端集。它被称为客户端集，因为它包含所有本地Kubernetes资源的多个客户端。

在群集中的pod中运行二进制文件时，`kubelet`将自动将服务帐户装入容器`/var/run/secrets/kubernetes.io/serviceaccount`。它替换刚刚提到的`kubeconfig`文件，可以很容易地`rest.Config`通过该`rest.InClusterConfig()`方法转换为。你经常会发现下列组合`rest.InClusterConfig()`并`clientcmd.BuildConfigFromFlags()`，包括支持`KUBECONFIG`环境变量：

```
config, err := rest.InClusterConfig()
if err != nil {
    // fallback to kubeconfig
    kubeconfig := filepath.Join("~", ".kube", "config")
    if envvar := os.Getenv("KUBECONFIG"); len(envvar) > 0 {
        kubeconfig = envvar
    }
    config, err = clientcmd.BuildConfigFromFlags("", kubeconfig)
    if err != nil {
        fmt.Printf("The kubeconfig cannot be loaded: %v\n", err)
        os.Exit(1)
    }
}
```

在下面的示例代码，我们在选择核心小组`v1`与`clientset.CoreV1()`再访问该吊舱“example”的“book”命名空间：

```
pod, err := clientset.CoreV1().Pods("book").Get("example", metav1.GetOptions{})
```

请注意，只有最后一个函数调用`Get`实际访问服务器。无论`CoreV1`和`Pods`选择客户，并只可用于以下设置的命名空间`Get`调用（这通常被称为所述生成器模式，在这种情况下建立的请求）。

该 Get 调用将HTTP GET 请求发送到服务器上的 `/api/v1/namespaces/book/pods/example`, 该请求在kubeconfig中设置。如果Kubernetes API服务器使用HTTP代码 200 进行应答, 则响应的主体将携带编码的pod对象, 或者作为 `client-go` JSON-作为协议缓冲区的默认有线格式。

注意

You 通过在其创建客户端之前修改REST配置, 可以为本机Kubernetes资源客户端启用protobuf:

```
cfg, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
cfg.AcceptContentTypes = "application/vnd.kubernetes.protobuf,
                           application/json"
cfg.ContentType = "application/vnd.kubernetes.protobuf"
clientset, err := kubernetes.NewForConfig(cfg)
```

请注意, 第4章中介绍的自定义资源不支持协议缓冲区。

版本控制和兼容性

Kubernetes API是版本。我们在上一节中看到过pods属于 `v1` 核心组。核心组实际上现在只存在于一个版本中。还有其他组, 例如, 该 `apps` 组, 存在于 `v1`, `v1beta2`, 和 `v1beta1` (撰写本文时)。如果您查看[k8s.io/api/apps](#)包, 您将找到这些版本的所有API对象。在[k8s.io/client-go/kubernetes/typed/apps](#)包中, 您将看到所有这些版本的客户端实现。

所有这些只是客户端。它没有说Kubernetes集群及其API服务器。使用具有API服务器不支持的API组版本的客户端将失败。客户端硬编码为某个版本, 应用程序开发人员必须选择正确的API组版本才能与手头的群集通信。有关API组兼容性保证的更多信息, 请参阅“[API版本和兼容性保证](#)”。

兼容性的第二个方面 `client-go` 是与之对话的API服务器的元API功能。例如, 有选项结构CRUD动词, 如 `CreateOptions`, `GetOptions`, `UpdateOptions`, 和 `DeleteOptions`。另一个重要的是 `ObjectMeta` (在[“ObjectMeta”](#)中详细讨论), 它是[各种类型](#)的一部分。所有这些都经常扩展到新功能; 我们通常称它们为API机械功能。在其字段的Go文档中, 注释指定功能何时被视为alpha或beta。相同的API兼容性保证适用于任何其他API字段。

在下面的示例中, `DeleteOptions` 结构在包[k8s.io/apimachinery/pkg/apis/meta/v1/types.go](#)中定义:

```
// DeleteOptions may be provided when deleting an API object.
type DeleteOptions struct {
    TypeMeta `json:",inline"`

    GracePeriodSeconds *int64 `json:"gracePeriodSeconds,omitempty"`
    Preconditions *Preconditions `json:"preconditions,omitempty"`
    OrphanDependents *bool `json:"orphanDependents,omitempty"`
    PropagationPolicy *DeletionPropagation `json:"propagationPolicy,omitempty"`

    // When present, indicates that modifications should not be
    // persisted. An invalid or unrecognized dryRun directive will
    // result in an error response and no further processing of the
    // request. Valid values are:
```

```
// - All: all dry run stages will be processed
// +optional
DryRun []string `json:"dryRun,omitempty" protobuf:"bytes,5,rep,name=dryRun"`
}
```

最后一个字段，`DryRun` 在Kubernetes 1.12中添加为alpha，在1.13中添加为beta（默认启用）。早期版本中的API服务器无法理解它。根据功能的不同，传递这样的选项可能会被忽略甚至被拒绝。因此，拥有一个`client-go` 与群集版本相距太远的版本非常重要。

小费

可以使用哪些字段的参考，其中质量级别是`k8s.io/api`中的源，例如，可以访问[release-1.13 分支中的 Kubernetes 1.13](#)。Alpha字段在其描述中标记为这样。

那里正在[生成API文档](#)，方便消费。但是，与`k8s.io/api`中的信息相同。

最后但并非最不重要的是，许多alpha和beta功能都有相应的[功能门](#)（请在此处查看[主要来源](#)）。功能在[问题中](#)被跟踪。

该集群和`client-go` 版本之间正式保证的支持矩阵在`client-go` `README`中发布（[参见表3-1](#)）。

	在州长 1.9	在州长 1.10	在州长 1.11	在州长 1.12	在州长 1.13	在州长 1.14	在州长 1.15
client-go 6.0	✓	+–	+–	+–	+–	+–	+–
client-go 7.0	+–	✓	+–	+–	+–	+–	+–
客户端去 8.0	+–	+–	✓	+–	+–	+–	+–
client-go 9.0	+–	+–	+–	✓	+–	+–	+–
client-go 10.0	+–	+–	+–	+–	✓	+–	+–
客户去 11.0	+–	+–	+–	+–	+–	✓	+–
client-go 12.0	+–	+–	+–	+–	+–	+–	✓
客户 - 去 HEAD	+–	+–	+–	+–	+–	+–	+–

- ✓：两者`client-go` 和Kubernetes版本具有相同的功能和相同的API组版本。
- +：`client-go` 具有Kubernetes群集中可能不存在的功能或API组版本。这可能是因为`client-go` Kubernetes删除了旧的，已弃用的功能，因此增加了功能。但是，它们共有的一切（即大多数API）都可以使用。
- –：`client-go` 明显与Kubernetes集群不兼容。

从外卖表3-1是该 client-go 库与它对应的集群版本的支持。在版本偏差的情况下，开发人员必须仔细考虑他们使用哪些功能和哪些API组，以及应用程序所说的群集版本是否支持这些功能。

在表3-1中，client-go 列出了版本。我们简单地提到“客户端库”即 client-go 使用语义版本化（semver）正式，但通过增加 client-go 每次的主要版本Kubernetes的次要版本（1.13.2中的13）增加。随着 client-go Kubernetes 1.4发布1.0，我们现在 client-go 为Kubernetes 1.15的12.0（在撰写本文时）。

此semver仅适用于 client-go 自身，而不适用于API Machinery或API存储库。相反，后者使用 Kubernetes版本进行标记，如图3-4所示。请参见“Vendoring”看看这是什么意思为vendoring k8s.io/client-go, k8s.io/apimachinery和k8s.io/api在您的项目。

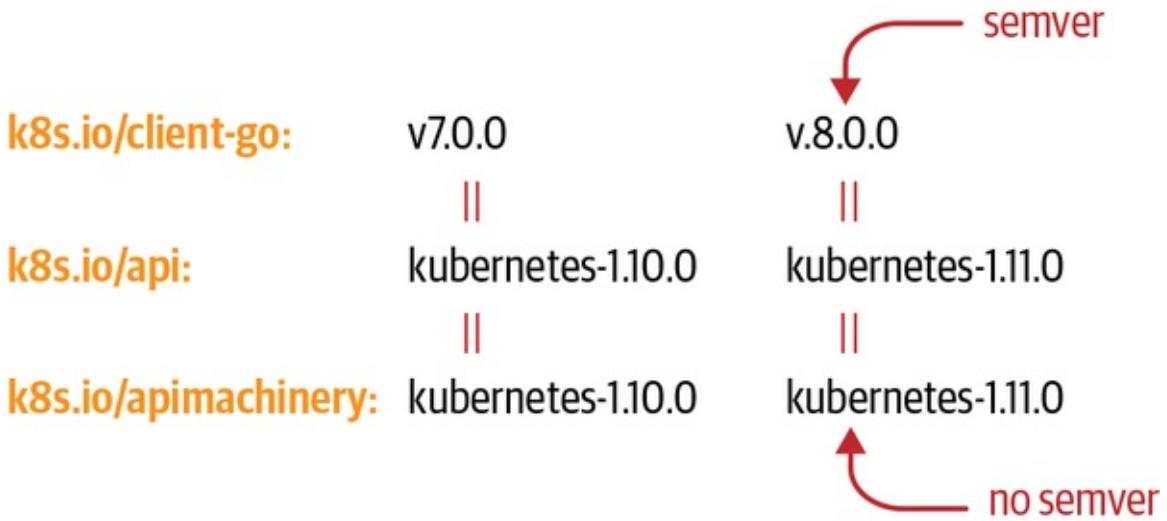


图3-4。客户端版本控制

API版本和兼容性保证

如果您使用代码定位不同的群集版本，那么在上一节中看到，选择正确的API组版本可能至关重要。Kubernetes版本所有API组。使用了一种常见的Kubernetes风格的版本控制方案，它由alpha, beta和GA（一般可用性）版本组成。

模式是：

- `v1alpha1`, `v1alpha2`, `v2alpha1`, 等有称为`alpha`版本并被认为是不稳定的 这意味着：
 - 他们可能会以任何不相容的方式随时离开或改变。
 - 从Kubernetes版本到版本，数据可能会被丢弃，丢失或无法访问。
 - 如果管理员未手动选择，则默认情况下通常会禁用它们。
- `v1beta1`, `v1beta2`, `v2beta1`, 等等，都称为`beta`版本。他们正在走向稳定，这意味着：
 - 对于至少一个Kubernetes版本，它们仍将与相应的稳定API版本并行存在。
 - 它们通常不会以不相容的方式改变，但没有严格的保证。
 - 存储在测试版中的对象不会被删除或无法访问。
 - 默认情况下，Beta版本通常在群集中启用。但这可能取决于所使用的Kubernetes分发或云提供商。

- `v1` , `v2` 等等是稳定的，通常可用的API; 那是：
 - 他们会留下来。
 - 它们兼容。

小费

Kubernetes有这些经验法则背后的[正式弃用政策](#)。您可以在[Kubernetes社区GitHub上](#)找到有关哪些API构造被认为兼容的更多详细信息。

与API组版本相关，要记住两个要点：

- API组版本作为一个整体应用于API资源，例如pod或服务的格式。除API组版本外，API资源可能具有正交版本的单个字段；例如，稳定API中的字段可能在其内联代码文档中标记为alpha质量。与刚刚为API组列出的规则相同的规则将适用于这些字段。例如：
 - 稳定API中的字段可能会变得不兼容，丢失数据或随时消失。例如，`ObjectMeta.Initializers` 从未提升过alpha 的字段将在不久的将来消失（在1.14中已弃用）：

```
// DEPRECATED - initializers are an alpha field and will be removed
// in v1.15.
Initializers *Initializers `json:"initializers,omitempty"
```

- 它通常在默认情况下处于禁用状态，必须使用API服务器功能门启用，如下所示：

```
type JobSpec struct {
    ...
    // This field is alpha-level and is only honored by servers that
    // enable the TTLAfterFinished feature.
    TTLSecondsAfterFinished *int32 `json:"ttlSecondsAfterFinished,omitempty"`
}
```

- API服务器的行为因字段而异。如果未启用相应的要素门，则会拒绝某些Alpha字段，并忽略某些字段。这在字段描述中有记录（参见 `TTLSecondsAfterFinished` 前面的示例）。
- 此外，API组版本在访问API时发挥作用。在同一资源的不同版本之间，有一个由API服务器完成的即时转换。也就是说，您可以 `v1beta1` 在任何其他受支持的版本（例如）中访问在一个版本（例如）中创建的对象，`v1` 而无需在您的应用程序中进行任何进一步的工作。这对于构建向后兼容和向前兼容的应用程序非常方便。
 - 存储的每个对象 `etcd` 都存储在特定版本中。通过默认情况下，这称为该资源的存储版本。虽然存储版本可以从Kubernetes版本更改为版本，但存储的对象在 `etcd` 撰写本文时不会自动更新。因此，在删除旧版本支持之前，集群管理员必须确保在更新Kubernetes集群时及时进行迁移。没有通用的迁移机制，迁移不同于Kubernetes分发到分发。
 - 但是，对于应用程序开发人员来说，这项操作工作根本不重要。即时转换将确保应用程序具有群集中对象的统一图片。应用程序甚至不会注意到正在使用哪个存储版本。存储版本控制

对于编写的Go代码是透明的。

Go中的Kubernetes对象

在“[创建和使用客户端](#)”中，我们了解了如何为核心组创建客户端以访问Kubernetes集群中的pod。在下文中，我们想要更详细地了解一下pod或任何其他Kubernetes资源，就此而言，就是Go的世界。

Kubernetes资源 - 或者更准确地说是对象 - 作为类型1的实例并且由API服务器作为资源提供，表示为结构。根据所涉及的种类，他们的领域当然不同。但另一方面，他们有一个共同的结构。

从类型系统的角度来看，Kubernetes对象实现了一个名为的Go接口 `runtime.Object` 从包`k8s.io/apimachinery/pkg/runtime`，这实际上非常简单：

```
// Object interface must be supported by all API types registered with Scheme.
// Since objects in a scheme are expected to be serialized to the wire, the
// interface an Object must provide to the Scheme allows serializers to set
// the kind, version, and group the object is represented as. An Object may
// choose to return a no-op ObjectKindAccessor in cases where it is not
// expected to be serialized.
type Object interface {
    GetObjectKind() schema.ObjectKind
    DeepCopyObject() Object
}
```

在这里，`schema.ObjectKind`（来自`k8s.io/apimachinery/pkg/runtime/schema`包）是另一个简单的界面：

```
// All objects that are serialized from a Scheme encode their type information.
// This interface is used by serialization to set type information from the
// Scheme onto the serialized version of an object. For objects that cannot
// be serialized or have unique requirements, this interface may be a no-op.
type ObjectKind interface {
    // SetGroupVersionKind sets or clears the intended serialized kind of an
    // object. Passing kind nil should clear the current setting.
    SetGroupVersionKind(kind GroupVersionKind)
    // GroupVersionKind returns the stored group, version, and kind of an
    // object, or nil if the object does not expose or provide these fields.
    GroupVersionKind() GroupVersionKind
}
```

换句话说，Go中的Kubernetes对象是一种数据结构，可以：

- 返回并设置组版儿童
- 被深刻复制

一个深拷贝是数据结构的克隆，使其不与原始对象共享任何内存。它用于代码必须在不修改原始对象的情况下改变对象的任何地方。有关如何在Kubernetes中实现深层复制的详细信息，请参阅有关代码生成的[全局标记](#)。

简而言之，对象存储其类型并允许克隆。

TypeMeta

虽然 `runtime.Object` 是只有一个界面，我们想知道它是如何实际实现的。来自 `k8s.io/api` 的 Kubernetes 对象 `schema.ObjectKind` 通过嵌入 `metav1.TypeMeta` 包 `k8s.io/apimachinery/pkg/apis/meta/v1` 中的结构来实现类型getter和setter：

```
// TypeMeta describes an individual object in an API response or request
// with strings representing the type of the object and its API schema version.
// Structures that are versioned or persisted should inline TypeMeta.
//
// +k8s:deepcopy-gen=false
type TypeMeta struct {
    // Kind is a string value representing the REST resource this object
    // represents. Servers may infer this from the endpoint the client submits
    // requests to.
    // Cannot be updated.
    // In CamelCase.
    // +optional
    Kind string `json:"kind,omitempty" protobuf:"bytes,1,opt,name=kind"`

    // APIVersion defines the versioned schema of this representation of an
    // object. Servers should convert recognized schemas to the latest internal
    // value, and may reject unrecognized values.
    // +optional
    APIVersion string `json:"apiVersion,omitempty"`
}
```

有了这个，Go中的pod声明如下所示：

```
// Pod is a collection of containers that can run on a host. This resource is
// created by clients and scheduled onto hosts.
type Pod struct {
    metav1.TypeMeta `json:",inline"`
    // Standard object's metadata.
    // +optional
    metav1.ObjectMeta `json:"metadata,omitempty"`

    // Specification of the desired behavior of the pod.
    // +optional
    Spec PodSpec `json:"spec,omitempty"`

    // Most recently observed status of the pod.
    // This data may not be up to date.
    // Populated by the system.
    // Read-only.
    // +optional
}
```

```
    Status PodStatus `json:"status,omitempty"`
}
```

如您所见，`TypeMeta` 嵌入式。此外，`pod`类型具有JSON标记，也声明 `TypeMeta` 为内联。

注意

这个`,inline` 标签实际上与Golang JSON en / decoders一起使用：嵌入式结构自动内联。

这在[YAML en / decoder go-yaml / yaml](#)中是不同的，它在非常早期的Kubernetes代码中与JSON并行使用。我们从那时起继承了[内联标签](#)，但今天它只是文档而没有任何影响。

在YAML串行foudn `k8s.io/apimachinery/pkg/runtime/serializer/yaml`使用`sigs.k8s.io/yaml`编组和取消编组功能。而这些依次编码和解码YAML `interface{}`，并使用JSON编码器进入Golang API结构并进行解码。

这匹配了一个pod的YAML表示，所有Kubernetes用户都知道：[2](#)

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: example
spec:
  containers:
  - name: hello
    image: debian:latest
    command:
    - /bin/sh
    args:
    - -c
    - echo "hello world"; sleep 10000
```

版本存储在 `TypeMeta.APIVersion`，种类中 `TypeMeta.Kind`。

核心小组因历史原因而异

类很早就添加到Kubernetes的许多其他类型都是核心组的一部分- 也称为遗留组 - 由空字符串表示。因此，`apiVersion` 只是设置为“`v1`”。

最终，API组被添加到Kubernetes中，并且以斜杠分隔的组名称被添加到 `apiVersion`。在这种情况下 `apps`，版本将是 `apps/v1`。因此，该 `apiVersion` 字段实际上是错误的名称；它存储API组名称和版本字符串。这是出于历史原因，因为 `apiVersion` 只有核心组才会定义，而这些其他API组都不存在。

运行[“创建和使用客户端”](#)中的示例以从群集中获取pod时，请注意客户端返回的pod对象实际上并未设置类型和版本集。`client-go` 基于in -ased的应用程序的惯例是这些字段在内存中是空的，只有当它们被封送到JSON或protobuf时，它们才会在线上填充实际值。然而，这是由客户端自动完成的，或者更准确地说，是由版本控制序列化器完成的。

幕后花絮：GO TYPE类型，包，种类和群组名称是如何相关的？

You可能想知道客户端如何知道填写该 `TypeMeta` 字段的种类和API组。虽然这个问题起初听起来微不足道，但它不是：

- 看起来这种类型只是Go类型名称，可以通过反射从对象派生。这基本上是正确的 - 可能在99%的情况下 - 但也有例外（在[第4章中](#)，您将了解自适应资源，但这不起作用）。
- 看起来该组只是Go包名称（`apps` API组的类型在[k8s.io/api/apps](#)中声明）。这通常匹配，但并非在所有情况下都匹配：核心组具有空组名称字符串，如我们所见。`rbac.authorization.k8s.io` 例如，组的类型位于[k8s.io/api/rbac](#)中，而不是[k8s.io/api/rbac.authorization.k8s.io](#)中。

如何填写该 `TypeMeta` 领域的问题的正确答案涉及方案的概念，将在[“方案”](#)中更详细地讨论。

换句话说，`client-go` 基于应用程序检查Golang类型的对象以确定手头的对象。这可能与其他框架不同，例如Operator SDK（请参阅[“Operator SDK”](#)）。

ObjectMeta

在此之外 `TypeMeta`，大多数顶级对象都有一个类型的字段 `metav1.ObjectMeta`，同样来自 `k8s.io/apimachinery/pkg/meta/v1`包：

```
type ObjectMeta struct {
    Name string `json:"name,omitempty"`
    Namespace string `json:"namespace,omitempty"`
    UID types.UID `json:"uid,omitempty"`
    ResourceVersion string `json:"resourceVersion,omitempty"`
    CreationTimestamp Time `json:"creationTimestamp,omitempty"`
    DeletionTimestamp *Time `json:"deletionTimestamp,omitempty"`
    Labels map[string]string `json:"labels,omitempty"`
    Annotations map[string]string `json:"annotations,omitempty"`
    ...
}
```

在JSON或YAML这些字段在元数据下。例如，对于上一个pod，`metav1.ObjectMeta` 存储：

```
metadata:
  namespace: default
  name: example
```

通常，它包含所有元级别信息，如名称，命名空间，资源版本（不要与API组版本混淆），多个时间戳，以及众所周知的标签和注释是其中的一部分 `objectMeta`。有关字段的深入讨论，请参阅[“类型剖析”](#) `ObjectMeta`。

前面在[“乐观并发”](#)中讨论了资源版本。它几乎不会从 `client-go` 代码中读取或写入。但它是 Kubernetes中的一个领域，使整个系统工作。`resourceVersion` 是的一部分 `ObjectMeta`，因为具有嵌入的每个对象 `ObjectMeta` 对应于一个键 `etcd` 的其中 `resourceVersion` 值起源。

规格和状态

最后，差不多了每个顶级对象都有一个 `spec` 和一个 `status` 部分。此约定来自Kubernetes API的声明性质：`spec` 用户需求，并且 `status` 是该愿望的结果，通常由系统中的控制器填充。有关 Kubernetes中控制器的详细讨论，请参阅[“控制器和操作员”](#)。

系统中的 `spec` 和 `status` 惯例只有少数例外- 例如，核心组中的端点，或RBAC对象 `ClusterRole` 。

客户端集

在在[“创建和使用客户端”](#)中的介绍性示例中，我们看到它 `kubernetes.NewForConfig(config)` 为我们提供了一个客户端集。客户端集提供对多个API组和资源的客户端的访问。在的情况 下 `kubernetes.NewForConfig(config)`，从`k8s.io/client-go/kubernetes`，我们送获得的中定义的所有API组和资源`k8s.io/api`。除了少数例外情况，例如 `APIServices` （对于聚合的API服务器）和 `CustomResourceDefinition` （参见[第4章](#)） - Kubernetes API服务器提供的整套资源。

在[第5章中](#)，我们将解释如何从API类型（在本例中为`k8s.io/api`）实际生成这些客户端集。使用自定义API的第三方项目不仅使用Kubernetes客户端集。所有客户端集的共同点是：REST配置（例如，返回者 `clientcmd.BuildConfigFromFlags("", *kubeconfig)`，如示例中所示）。

该客户端设置`k8s.io/client-go/kubernetes/typed`中的主界面，*Kubernetes*本地资源如下所示：

```
type Interface interface {
    Discovery() discovery.DiscoveryInterface
    AppsV1() appsv1.AppsV1Interface
    AppsV1beta1() appsv1beta1.AppsV1beta1Interface
    AppsV1beta2() appsv1beta2.AppsV1beta2Interface
    AuthenticationV1() authenticationv1.AuthenticationV1Interface
    AuthenticationV1beta1() authenticationv1beta1.AuthenticationV1beta1Interface
    AuthorizationV1() authorizationv1.AuthorizationV1Interface
    AuthorizationV1beta1() authorizationv1beta1.AuthorizationV1beta1Interface

    ...
}
```

在这个界面中曾经有过无版本的方法 - 例如，`Apps()` `appsv1.AppsV1Interface` 但是从基于 Kubernetes 1.14的 `client-go` 11.0 开始，它们被弃用了。如前所述，对应用程序使用的API组的版本非常明确是一种很好的做法。

过去版本化的客户和内部客户

在过去，Kubernetes有所谓的内部客户。这些对象称为“内部”的对象使用了通用的内存版本，其中包含与线上版本之间的转换。

希望是从正在使用的实际API版本中抽象出控制器代码，并能够通过单行更改切换到另一个版本。实际上，实现转换的巨大额外复杂性以及此转换代码所需的有关对象语义的知识量导致了这种模式不值得的结论。

此外，客户端和API服务器之间从未进行任何类型的自动协商。即使使用内部类型和客户端，控制器也硬编码到线路上的特定版本。因此，对于使用版本化API类型的客户端和服务器之间的版本偏差，使用内部类型的控制器不再兼容。

在最近的Kubernetes版本中，大量代码被重写以完全摆脱这些内部版本。今天，有没有内部版本`k8s.io/api`可用，也没有客户`k8s.io/client-go`。

一切客户端集还允许访问发现客户端（它将被使用`RESTMappers`；请参阅“[REST映射](#)”和“[从命令行使用API](#)”）。

背后每个`GroupVersion`方法（例如`AppsV1beta1`），我们找到API组的资源 - 例如：

```
type AppsV1beta1Interface interface {
    RESTClient() rest.Interface
    ControllerRevisionsGetter
    DeploymentsGetter
    StatefulSetsGetter
}
```

同`RESTClient`作为通用`REST`客户端，每个资源有一个接口，如：

```
// DeploymentsGetter has a method to return a DeploymentInterface.
// A group's client should implement this interface.
type DeploymentsGetter interface {
    Deployments(namespace string) DeploymentInterface
}

// DeploymentInterface has methods to work with Deployment resources.
type DeploymentInterface interface {
    Create(*v1beta1.Deployment) (*v1beta1.Deployment, error)
    Update(*v1beta1.Deployment) (*v1beta1.Deployment, error)
    UpdateStatus(*v1beta1.Deployment) (*v1beta1.Deployment, error)
    Delete(name string, options *v1.DeleteOptions) error
    DeleteCollection(options *v1.DeleteOptions, listOptions v1.ListOptions) error
    Get(name string, options v1.GetOptions) (*v1beta1.Deployment, error)
    List(opts v1.ListOptions) (*v1beta1.DeploymentList, error)
    Watch(opts v1.ListOptions) (watch.Interface, error)
    Patch(name string, pt types.PatchType, data []byte, subresources ...string)
        (result *v1beta1.Deployment, err error)
    DeploymentExpansion
}
```

根据资源的范围 - 即，是集群还是命名空间作用域 - 访问者（此处`DeploymentGetter`）可能有也可能没有`namespace`参数。

在 `DeploymentInterface` 可以访问资源的所有受支持的动词。其中大多数是不言自明的，但接下来将描述需要额外评论的那些。

状态子资源：`UpdateStatus`

部署有一个所谓状态子资源。这意味着 `UpdateStatus` 使用后缀的额外HTTP端点 `/status`。虽然 `apis / apps / v1beta1 / namespaces / ns/ deployments /name` endpoint 上的更新只能更改部署的规范，但端点 `apis / apps / v1beta1 / namespaces / ns/ deployments name/ status` 只能更改对象的状态。这对于为规范更新（由人完成）和状态更新（由控制器完成）设置不同的权限非常有用。

通过默认 `client-gen`（参见“[client-gen Tags](#)”）生成 `UpdateStatus()` 方法。该方法的存在并不能保证资源实际上支持子资源。当我们在“[子资源](#)”中使用CRD时，这将非常重要。

列表和删除

`DeleteCollection` 允许我们一次删除命名空间的多个对象。该 `ListOptions` 参数允许我们定义应删除哪些对象使用字段或标签选择器：

```
type ListOptions struct {
    ...
    // A selector to restrict the list of returned objects by their labels.
    // Defaults to everything.
    // +optional
    LabelSelector string `json:"labelSelector,omitempty"`
    // A selector to restrict the list of returned objects by their fields.
    // Defaults to everything.
    // +optional
    FieldSelector string `json:"fieldSelector,omitempty"`
    ...
}
```

手表

`watch` 给一个事件接口，用于对象的所有更改（添加，删除和更新）。`watch.Interface` 从 `k8s.io/apimachinery/pkg/watch` 返回的内容如下所示：

```
// Interface can be implemented by anything that knows how to watch and
// report changes.
type Interface interface {
    // Stops watching. Will close the channel returned by ResultChan(). Releases
    // any resources used by the watch.
    Stop()

    // Returns a chan which will receive all the events. If an error occurs
    // in the watch loop, the channel will be closed.
```

```
// or Stop() is called, this channel will be closed, in which case the
// watch should be completely cleaned up.
ResultChan() <-chan Event
}
```

`watch` 接口的结果通道返回三种事件：

```
// EventType defines the possible types of events.
type EventType string

const (
    Added     EventType = "ADDED"
    Modified  EventType = "MODIFIED"
    Deleted   EventType = "DELETED"
    Error     EventType = "ERROR"
)

// Event represents a single event to a watched resource.
// +k8s:deepcopy-gen=true
type Event struct {
    Type EventType

    // Object is:
    // * If Type is Added or Modified: the new state of the object.
    // * If Type is Deleted: the state of the object immediately before
    // deletion.
    // * If Type is Error: *api.Status is recommended; other types may
    // make sense depending on context.
    Object runtime.Object
}
```

虽然直接使用这个界面很诱人，但实际上它实际上不鼓励有利于线人（参见“[线人和缓存](#)”）。

Informers是此事件接口和带有索引查找的内存缓存的组合。这是迄今为止最常见的手表用例。在引擎线下，首先调用 `List` 客户端获取所有对象的集合（作为缓存的基线），然后 `watch` 更新缓存。它们正确处理错误情况 - 即从网络问题或其他群集问题中恢复。

客户扩展

`DeploymentExpansion` 是实际上是一个空接口。它用于添加自定义客户端行为，但现在很难用于 Kubernetes。相反，客户端生成器允许我们以声明方式添加自定义方法（请参阅[“client-gen标签”](#)）。

再次注意，所有的这些方法 `DeploymentInterface` 既不期望在有效信息 `TypeMeta` 的字段 `Kind` 和 `APIVersion`，也没有设定这些字段 `Get()` 和 `List()`（见[“TypeMeta”](#)）。这些字段仅在线路上填充实际值。

客户选项

它值得看看我们在创建客户端集时可以设置的不同选项。在“[版本控制和兼容性](#)”之前的注释中，我们看到我们可以切换到原生Kubernetes类型的protobuf线格式。Protobuf比JSON（空间和客户端和服务器的CPU负载）更有效，因此更受欢迎。

出于调试目的和指标的可读性，区分访问API服务器的不同客户端通常很有帮助。至这样做，我们可以在REST配置中设置用户代理字段。默认值为 `binary/version (os/arch) kubernetes/commit`；例如，`kubectl` 将使用像这样的用户代理 `kubectl/v1.14.0 (darwin/amd64) kubernetes/d654b49`。如果该模式不足以进行设置，则可以像这样定制：

```
cfg, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
cfg.AcceptContentTypes = "application/vnd.kubernetes.protobuf,application/json"
cfg.UserAgent = fmt.Sprintf(
    "book-example/v1.0 (%s/%s) kubernetes/v1.0",
    runtime.GOOS, runtime.GOARCH
)
clientset, err := kubernetes.NewForConfig(cfg)
```

其他REST配置中经常被覆盖的值是客户端速率限制和超时的值：

```
// Config holds the common attributes that can be passed to a Kubernetes
// client on initialization.
type Config struct {
    ...
    // QPS indicates the maximum QPS to the master from this client.
    // If it's zero, the created RESTClient will use DefaultQPS: 5
    QPS float32
    ...
    // Maximum burst for throttle.
    // If it's zero, the created RESTClient will use DefaultBurst: 10.
    Burst int
    ...
    // The maximum length of time to wait before giving up on a server request.
    // A value of zero means no timeout.
    Timeout time.Duration
    ...
}
```

该 `QPS` 值默认为 5 请求每秒，有一阵爆冷 10。

该`timeout`没有默认值，至少在客户端REST配置中没有。默认情况下，Kubernetes API服务器将在60秒后超时不是长时间运行请求的每个请求。长时间运行的请求可以是监视请求，也可以是对/`exec`，/`portforward`或/`proxy`等子资源的无限请求。

优雅的关机和对连接错误的适应能力

要求分为长期运行和非长期运行。手表是长时间运行，同时 `GET` , `LIST` , `UPDATE` 等都是非长期运行。许多子资源（例如，用于日志流，执行，端口转发）也是长期运行的。

重新启动Kubernetes API服务器时（例如，在更新期间），它会等待最多60秒才能正常关闭。在此期间，它完成非长时间运行的请求，然后终止。当它终止时，长时间运行的请求（如正在进行的监视连接）将被切断。

非长时间运行的请求无论如何都会受到60秒的限制（然后它们会超时）。因此，从客户端的角度来看，关闭是优雅的。

通常，应始终为不成功的请求准备应用程序代码，并且应该以对应用程序不致命的方式进行响应。在分布式系统的世界中，那些连接错误是正常的，没什么好担心的。但需要特别注意小心处理错误情况并从中恢复。

错误处理尤为重要的手表。手表长时间运行，但它们可能随时出现故障。下一节中描述的告密者提供了一个围绕监视的弹性实现，并且可以优雅地处理错误 - 也就是说，它们可以通过新连接从断开连接中恢复。应用程序代码通常不会注意到。

告密者和缓存

该“客户端集”中的客户端接口包括 `Watch` 动词，动词提供对对象的更改（添加，删除，更新）作出反应的事件接口。Informers为最常见的手表用例提供了更高级别的编程接口：内存中缓存和按名称或内存中的其他属性快速，索引查找对象。

每次需要对象时访问API服务器的控制器都会在系统上产生高负载。使用informers进行内存中缓存是解决此问题的方法。此外，线人几乎可以实时响应对象的变化，而不需要轮询请求。

[图3-5](#)显示了线人的概念模式；具体来说，他们：

- 从API服务器获取输入作为事件。
- 提供一个类似客户端的接口，`Lister` 用于从内存缓存中获取和列出对象。
- 注册事件处理程序以进行添加，删除和更新。
- 实现内存缓存使用商店。

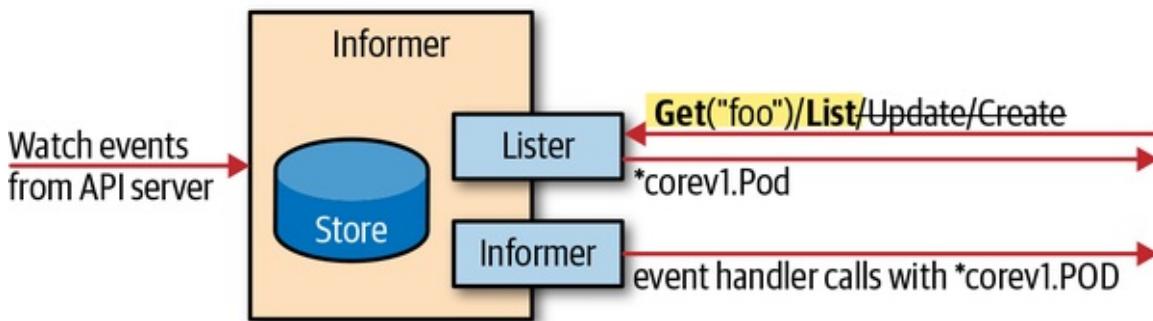


图3-5。线人

线人还有高级错误行为：当长时间运行的监视连接发生故障时，它们会通过尝试另一个监视请求从中恢复，从而在不丢失任何事件的情况下拾取事件流。如果中断很长，并且API服务器丢失了事件，因为 `etcd` 在新的监视请求成功之前将它们从数据库中清除，则告密者将重新安置所有物体。

在`relists`旁边，有一个可配置的重新同步周期，用于内存缓存和业务逻辑之间的协调：每次经过此周期后，将为所有对象调用已注册的事件处理程序。常见值以分钟为单位（例如，10或30分钟）。

警告

重新同步纯粹是在内存中，不会触发对服务器的调用。这曾经是不同的，但[最终被改变了](#)，因为手表机制的错误行为已经得到了足够的改进，使得不需要重复。

所有这些高级且经过实战验证的错误行为是使用informer而不是`Watch()`直接使用客户端方法推出自定义逻辑的一个很好的理由。Informer在Kubernetes本身随处可见，是Kubernetes API设计中的主要架构概念之一。

虽然告密者比轮询更受欢迎，但它们会在API服务器上产生负载。一个二进制文件应该只为每个`GroupVersionResource`实例化一个informer。至通过使用共享的`informer`工厂，我们可以实现一个告密者。

共享的informer工厂允许在应用程序中为相同的资源共享信息。换句话说，不同的控制回路可以使用与引擎盖下的API服务器相同的手表连接。对于例如，`kube-controller-manager`其中一个主要的Kubernetes集群组件（参见[“API服务器”](#)）具有更大的两位数控制器。但是对于每个资源（例如，`pod`），在该过程中只有一个告密者。

小费

始终使用共享的informer工厂来实例化informers。不要尝试手动实例化informers。开销很小，并且不使用共享信息器的非平凡控制器二进制文件可能正在为某个地方的同一资源打开多个监视连接。

从一个开始REST配置（请参阅[“创建和使用客户端”](#)），使用客户端集创建共享的informer工厂很容易。告密者由代码生成器生成，并作为`client-go k8s.io/client-go/informers`中标准Kubernetes资源的一部分提供：

```
import (
    ...
    "k8s.io/client-go/informers"
)

...
clientset, err := kubernetes.NewForConfig(config)
informerFactory := informers.NewSharedInformerFactory(clientset, time.Second*30)
podInformer := informerFactory.Core().V1().Pods()
podInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc: func(new interface{}) {...},
    UpdateFunc: func(old, new interface{}) {...},
    DeleteFunc: func(obj interface{}) {...},
})
informerFactory.Start(wait.NeverStop)
informerFactory.WaitForCacheSync(wait.NeverStop)
pod, err := podInformer.Lister().Pods("programming-kubernetes").Get("client-go")
```

该示例显示了如何获取pod的共享informer。

你可以看到informers允许添加三种情况的事件处理程序添加，更新和删除。这些通常用于触发控制器的业务逻辑- 即再次处理某个对象（请参阅[“控制器和操作员”](#)）。这些处理程序通常只是将修改后的对象添加到工作队列中。

其他事件处理程序和内部存储更新逻辑

不要将这些处理程序与informer内部存储库的更新逻辑（可通过示例最后一行中的lister访问）混淆。告密者将始终更新其商店，但刚刚描述的其他事件处理程序是可选的，并且意味着由线人的消费者使用。

另请注意，可以添加许多事件处理程序。存在整个共享信息工厂概念只是因为这在具有许多控制循环的控制器二进制文件中非常常见，每个控制循环都安装事件处理程序以将对象添加到它们自己的工作队列中。

注册处理程序后，必须启动共享的线程工厂。在引擎盖下有Go例程来执行对API服务器的实际调用。该 `Start` 方法（使用停止通道来控制生命周期）启动这些Go例程，该 `WaitForCacheSync()` 方法使代码等待对 `List` 客户端的第一次调用完成。如果控制器逻辑要求填充缓存，则此 `WaitForCacheSync` 调用至关重要。

通常，幕后手表的事件界面导致一定的滞后。在具有适当容量规划的设置中，这种滞后并不是很大。当然，使用指标衡量这种滞后是一种很好的做法。但是滞后存在，所以应用程序逻辑的构建方式使得滞后不会损害代码的行为。

警告

告密者的滞后可能导致控制器 `client-go` 直接在API服务器上进行的更改与告密者所知的世界状态之间的竞争。

如果控制器更改了对象，则同一进程中的告密者必须等到相应的事件到达，然后更新内存中的存储。此过程不是即时的，并且可能在前一个更改变为可见之前通过另一个触发器启动另一个控制器工作循环运行。

在该示例中，重新同步间隔30秒导致发送到注册的完整事件集 `UpdateFunc`，使得控制器逻辑能够将其状态与API服务器的状态协调。通过比较该 `ObjectMeta.resourceVersion` 字段，可以区分真实更新和重新同步。

小费

选择良好的重新同步间隔取决于上下文。例如，30秒非常短。在许多情况下，几分钟，甚至30分钟，是一个不错的选择。在最坏的情况下，30分钟意味着需要30分钟才能通过协调修复代码中的错误（例如，由于错误处理错误导致的信号丢失）。

另请注意，示例调用中的最后一行 `Get("client-go")` 纯粹是在内存中；无法访问API服务器。无法直接修改内存存储中的对象。相反，客户端集必须用于对资源的任何写访问。然后，线程将从API服务器获取事件并更新其内存存储。

永远不要改变告密者的对象

记住从列表传递给事件处理程序的任何对象都是由线人拥有的，这一点非常重要。如果你以任何方式改变它，你就冒着引入难以调试的风险将一致性问题缓存到您的应用程序中。在更改对象之前，请始终执行深层复制（请参阅[“Go中的Kubernetes对象”](#)）。

一般来说：在变异对象之前，总是问自己谁拥有这个对象或其中的数据结构。根据经验：

- 告密者和姐妹们拥有他们返回的物品。因此，消费者必须在变异之前进行深层复制。
- 客户端返回呼叫者拥有的新鲜对象。
- 转换返回共享对象。如果调用者拥有输入对象，则它不拥有输出。

`NewSharedInformerFactory` 示例中的`informer`构造函数将资源的所有对象缓存在商店的所有名称空间中。如果这对于应用程序来说太多了，那么有一个具有更大灵活性的替代构造函数：

```
// NewFilteredSharedInformerFactory constructs a new instance of
// sharedInformerFactory. Listers obtained via this sharedInformerFactory will be
// subject to the same filters as specified here.
func NewFilteredSharedInformerFactory(
    client versioned.Interface, defaultResync time.Duration,
    namespace string,
    tweakListOptions internalinterfaces.TweakListOptionsFunc
) SharedInformerFactory

type TweakListOptionsFunc func(*v1.ListOptions)
```

它使我们能够指定一个命名空间，并传递一个 `TweakListOptionsFunc`，这可能发生变异的 `ListOptions` 用来列出并观看使用对象结构 `List` 和 `watch` 客户端的调用。例如，它可用于设置标签或字段选择器。

告密者是控制器的基石之一。在[第6章中](#)，我们将看到 `client-go` 基于典型的控制器的外观。在客户端和线人之后，第三个主要构建块是工作队列。我们现在来看看吧。

工作队列

一个工作队列是一种数据结构。您可以按队列预定义的顺序添加元素并从队列中取出元素。形式上，这种队列称为优先级队列。`client-go` 为在k8s.io/client-go/util/workqueue中构建控制器提供了一个强大的实现。

更准确地说，该包装包含许多用于不同目的的变体。所有变体实现的基本接口如下所示：

```
type Interface interface {
    Add(item interface{})
    Len() int
    Get() (item interface{}, shutdown bool)
    Done(item interface{})
    ShutDown()
    ShuttingDown() bool
}
```

这里 `Add(item)` 添加一个项目，`Len()` 给出长度，并 `Get()` 返回一个具有最高优先级的项目（并且它会阻塞直到一个可用）。当控制器完成处理后，返回的每个项目都 `Get()` 需要 `Done(item)` 调用。同时，重复 `Add(item)` 只会将项目标记为脏，以便在 `Done(item)` 被调用时对其进行读取。

以下队列类型派生自此通用接口：

- `DelayingInterface` 可以在以后添加项目。这样可以更容易在故障后重新排队项目而不会在热循环中结束：

```
type DelayingInterface interface {
    Interface
    // AddAfter adds an item to the workqueue after the
    // indicated duration has passed.
    AddAfter(item interface{}, duration time.Duration)
}
```

- `RateLimitingInterface` 速率限制项目被添加到队列中。它扩展了 `DelayingInterface`：

```
type RateLimitingInterface interface {
    DelayingInterface

    // AddRateLimited adds an item to the workqueue after the rate
    // limiter says it's OK.
    AddRateLimited(item interface{})

    // Forget indicates that an item is finished being retried.
    // It doesn't matter whether it's for perm failing or success;
    // we'll stop the rate limiter from tracking it. This only clears
    // the `rateLimiter`; you still have to call `Done` on the queue.
    Forget(item interface{})

    // NumRequeues returns back how many times the item was requeued.
    NumRequeues(item interface{}) int
}
```

这里最有趣的是 `Forget(item)` 方法：它重置给定项目的后退。通常，在成功处理项目时将调用它。

速率限制算法可以传递给构造函数 `NewRateLimitingQueue`。在同一个包中定义了几个速率限制器，例如 `BucketRateLimiter`，`the ItemExponentialFailureRateLimiter`，`the ItemFastSlowRateLimiter` 和 `the MaxOfRateLimiter`。有关更多详细信息，请参阅包文档。大多数控制器只使用这些 `DefaultControllerRateLimiter() *RateLimiter` 功能，这给出了：

- 指数退避从5毫秒开始并上升到1,000秒，使每个错误的延迟加倍
- 最大速率为每秒10项，爆发100项

根据上下文，您可能希望自定义值。对于某些控制器应用，每个项目最大后退1,000秒是很多的。

API机械深度

该API Machinery存储库实现了Kubernetes类型系统的基础知识。但究竟什么是这种类型的系统呢？什么是开始的类型？

该术语类型实际上并不存在于API Machinery的术语中。相反，它指的是种类。

种

种分为API组和版本，正如我们在“[API术语](#)”中已经看到的那样。因此，API Machinery存储库中的核心术语是GroupVersionKind，简称GVK。

在Go中，每个GVK对应一个Go类型。相反，Go类型可以属于多个GVK。

种类没有正式映射到HTTP路径一对一。许多种类都有HTTP REST端点，用于访问给定类型的对象。但是也有没有任何HTTP端点的种类（例如，[admission.k8s.io/v1beta1.AdmissionReview](#)，用于调用webhook）。还有许多端点返回的类型 - 例如，[meta.k8s.io/v1.Status](#)，由所有端点返回以报告非对象状态，如错误。

通过约定，种类在[CamelCase](#)中被格式化为单词并且通常是单数的。根据具体情况，它们的具体格式不同。对于CustomResourceDefinition类型，它必须是DNS路径标签（RFC 1035）。

资源

在与“种类”并行，正如我们在“[API术语](#)”中看到的那样，存在资源的概念。资源再次分组和版本化，导致术语GroupVersionResource或简称GVR。

每个GVR对应一个HTTP（基本）路径。GVR用于标识Kubernetes API的REST端点。例如，GVR `apps / v1.deployments` 映射到 `/apis / apps / v1 / namespaces / namespace/ deployments`。

客户端库使用此映射来构造访问GVR的HTTP路径。

了解资源是命名空间还是群集范围

You必须知道GVR是命名空间还是群集范围才能知道HTTP路径。例如，部署是命名空间，因此将命名空间作为其HTTP路径的一部分。其他GVR，例如[rbac.authorization.k8s.io/v1.clusterroles](#)，是集群范围的；例如，可以在[apis / rbac.authorization.k8s.io / v1 / clusterroles](#)中访问集群角色。

通过惯例，资源是小写和复数，通常对应于并行类型的多个单词。它们必须符合DNS路径标签格式（RFC 1025）。由于资源直接映射到HTTP路径，因此这并不奇怪。

REST映射

该将GVK映射到GVR称为REST映射。

A `RESTMapper` 是[Golang接口](#)，使我们能够为GVK请求GVR：

```
RESTMapping(gk schema.GroupKind, versions ...string) (*RESTMapping, error)
```

RESTMapping 右侧的类型如下所示：

```
type RESTMapping struct {
    // Resource is the GroupVersionResource (location) for this endpoint.
    Resource schema.GroupVersionResource.

    // GroupVersionKind is the GroupVersionKind (data format) to submit
    // to this endpoint.
    GroupVersionKind schema.GroupVersionKind

    // Scope contains the information needed to deal with REST Resources
    // that are in a resource hierarchy.
    Scope RESTScope
}
```

此外，a RESTMapper 还提供了许多便利功能：

```
// KindFor takes a partial resource and returns the single match.
// Returns an error if there are multiple matches.
KindFor(resource schema.GroupVersionResource) (schema.GroupVersionKind, error)

// KindsFor takes a partial resource and returns the list of potential
// kinds in priority order.
KindsFor(resource schema.GroupVersionResource) ([]schema.GroupVersionKind, error)

// ResourceFor takes a partial resource and returns the single match.
// Returns an error if there are multiple matches.
ResourceFor(input schema.GroupVersionResource) (schema.GroupVersionResource, error)

// ResourcesFor takes a partial resource and returns the list of potential
// resource in priority order.
ResourcesFor(input schema.GroupVersionResource) ([]schema.GroupVersionResource, error)

// RESTMappings returns all resource mappings for the provided group kind
// if no version search is provided. Otherwise identifies a preferred resource
// mapping for the provided version(s).
RESTMappings(gk schema.GroupKind, versions ...string) ([]*RESTMapping, error)
```

这里，部分GVR意味着并非所有字段都被设置。例如，假设您输入 `**kubectl get pods**`。在这种情况下，组和版本丢失。一个 RESTMapper 有足够的信息可能仍然设法将其映射到 v1 Pods 的那种。

对于前面的部署示例， RESTMapper 了解部署（更多关于这意味着什么）将把 `apps / v1.Deployment` 映射到 `apps / v1.deployments` 作为命名空间资源。

RESTMapper 接口有多种不同的实现方式。客户端应用程序最重要的一个基于发现的 `k8s.io/client-go/restmapper` `DeferredDiscoveryRESTMapper` 包：它使用来自Kubernetes API服务器的发现信息来动态构建REST映射。它还可以与非核心资源（如自定义资源）一起使用。

方案

该我们希望在座的Kubernetes类型系统的情况下最终核心概念是计划在包 `k8s.io/apimachinery/pkg/runtime`。

一个方案将Golang世界与GVKs独立于实现的世界联系起来。方案的主要特征是将Golang类型映射到可能的GVK：

```
func (s *Scheme) ObjectKinds(obj Object) ([]schema.GroupVersionKind, bool, error)
```

正如我们在看到“围棋Kubernetes对象”，一个objec t可返回其组，通过T个样，他 `GetObjectKind()` `schema.ObjectKind` 的方法。但是，这些值在大多数时间都是空的，因此对于识别而言是无用的。

相反，该方案采用给定对象的Golang类型反射并将其映射到Golang类型的已注册GVK。当然，要使用它，Golang类型必须像这样注册到方案中：

```
scheme.AddKnownTypes(schema.GroupVersionKind{}, "v1", "Pod", &Pod{})
```

该方案不仅用于注册Golang类型及其GVK，还用于存储转换函数和默认值的列表（参见图3-6）。我们将在第8章中更详细地讨论转换和违约者。它也是实现编码器和解码器的数据源。

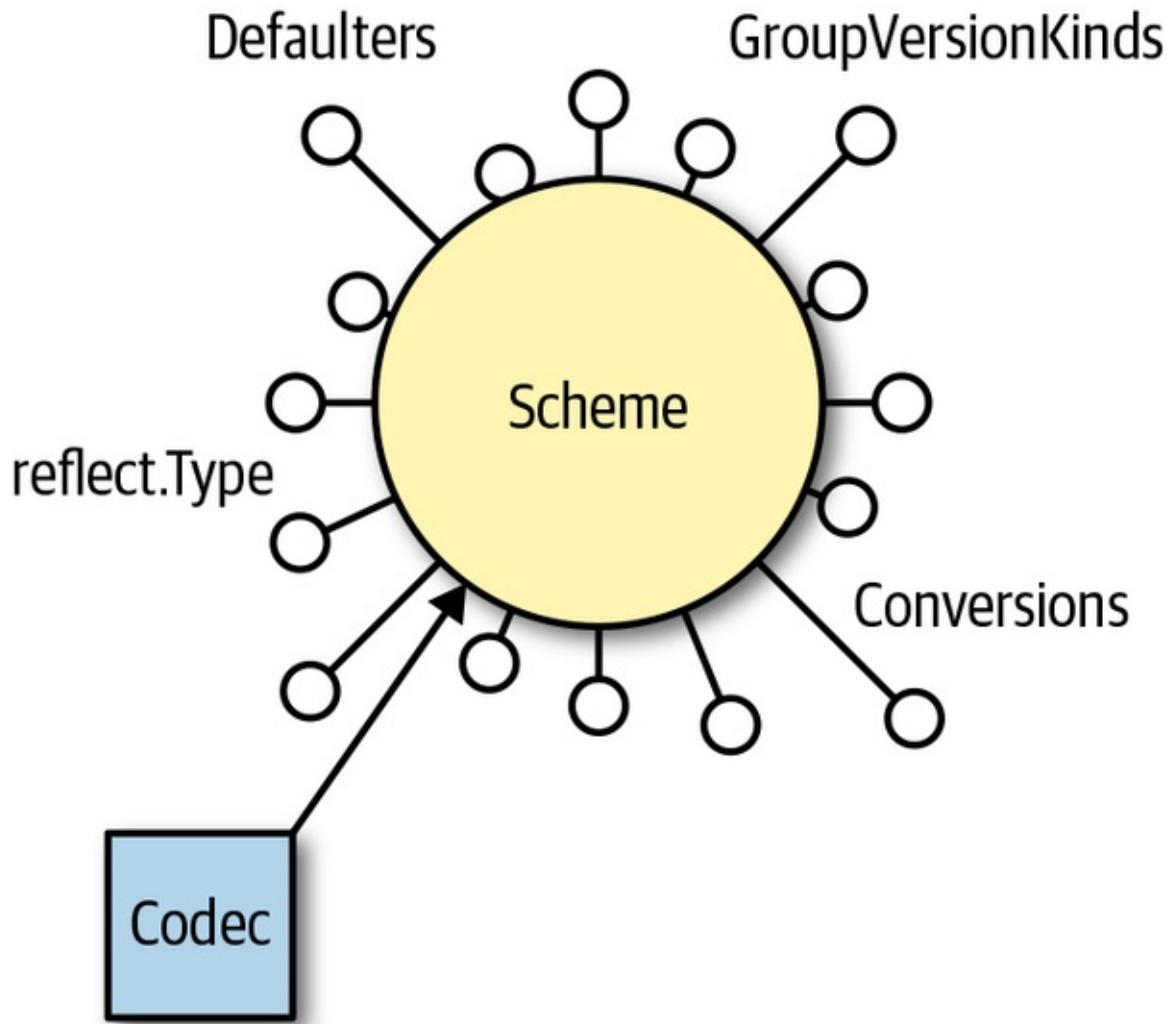


图3-6。该方案将Golang数据类型与GVK，转换和违约者联系起来

对于Kubernetes核心类型，在`k8s.io/client-go/kubernetes/scheme`包中的 `client-go` 客户端集中有一个[预定义的方案](#)，所有类型都已预先注册。实际上，每个客户端集都由 `client-gen` 代码生成器（参见第5章）具有子包，`scheme` 其中包含客户端集中所有组和版本中的所有类型。

同该计划我们深入探讨了API Machinery的概念。如果你只记得关于这些概念的一件事，那就让它如图3-7所示。

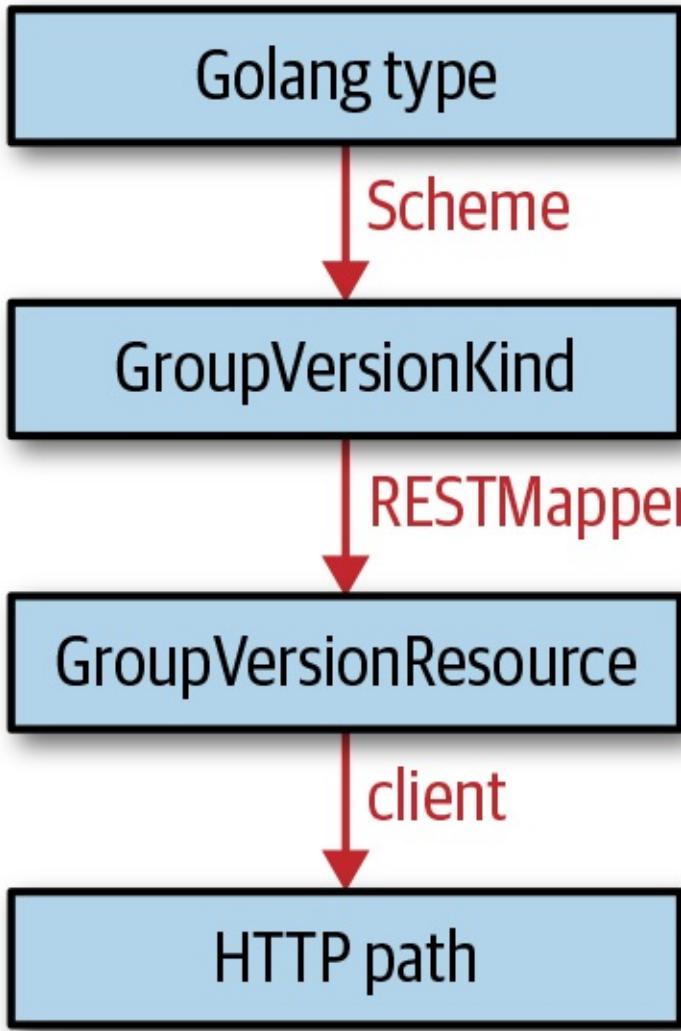


图3-7。从Golang类型到GVK再到GVR再到HTTP路径 - 简而言之

Vendoring

我们本章在看到 `k8s.io/client-go`, `k8s.io/api` 和 `k8s.io/apimachinery` 是中央在 Golang Kubernetes 编程。Golang 采用 *vendoring* 包括在第三方应用程序的源代码库，这些库。

Vendoring 是 Golang 社区的一个移动目标。在撰写本文时，几种销售工具很常见，例如 `godeps`, `dep` 和 `glide`。与此同时，Go 1.12 正在获得对 Go 模块的支持，这些模块可能会成为未来 Go 社区的标准销售方法，但目前尚未在 Kubernetes 生态系统中做好准备。

现在大多数项目都使用 `dep` 或 `glide`。Kubernetes 本身在 `github.com/kubernetes/kubernetes` 中跳转到了 1.15 开发周期的 Go 模块。以下注释与所有这些销售工具相关。

每个 `k8s.io/**` 存储库中受支持的依赖项版本的真实来源是随附的 `Godeps / Godeps.json*` 文件。重要的是要强调任何其他依赖项选择都可能破坏库的功能。

有关 `k8s.io/client-go`, `k8s.io/api` 和 `k8s.io/apimachinery` 的已发布标记以及哪些标记彼此兼容，请参阅“[客户端库](#)”。

滑行

项目using `glide` 可以使用它在任何依赖项更改时读取*Godeps / Godeps.json*文件的能力。事实证明这非常可靠：开发人员只需要声明正确的*k8s.io/client-go*版本，并 `glide` 选择正确版本的*k8s.io/apimachinery*, *k8s.io / api*和其他依赖项。

对于GitHub上的一些项目，`glide.yaml`文件可能如下所示：

```
package: github.com/book/example
import:
- package: k8s.io/client-go
  version: v10.0.0
...
```

有了它，`glide install -v` 将*k8s.io/client-go*及其依赖项下载到本地供应商/包中。这里，`-v` 意味着从销售的库中删除供应商/包。这是我们的目的所必需的。

如果您 `client-go` 通过编辑`glide.yaml`更新到新版本，`glide update -v` 将再次以正确的版本下载新的依赖项。

DEP

`dep` 通常被认为比...更强大和更先进 `glide`。长期以来它被视为继任者 `glide` 的生态系统，似乎注定要在围棋vendoring工具。在撰写本文时，其未来尚不清楚，Go模块似乎是前进的道路。

在上下文中 `client-go`，了解以下几个限制非常重要 `dep`：

- `dep` 在第一次运行时会读取*Godeps / Godeps.json* `dep init`。
- `dep` 在以后的调用中不会读取*Godeps / Godeps.json* `dep ensure -update`。

这意味着在*Godep.toml*中更新版本 `client-go` 时，依赖关系的解析可能是错误的。这很不幸，因为它要求开发人员显式地并且通常手动声明所有依赖项。`client-go`

一个工作且一致的*Godep.toml*文件如下所示：

```
[[constraint]]
name = "k8s.io/api"
version = "kubernetes-1.13.0"

[[constraint]]
name = "k8s.io/apimachinery"
version = "kubernetes-1.13.0"

[[constraint]]
name = "k8s.io/client-go"
version = "10.0.0"

[prune]
```

```

go-tests = true
unused-packages = true

# the following overrides are necessary to enforce
# the given version, even though our
# code does not import the packages directly.
[[override]]
name = "k8s.io/api"
version = "kubernetes-1.13.0"

[[override]]
name = "k8s.io/apimachinery"
version = "kubernetes-1.13.0"

[[override]]
name = "k8s.io/client-go"
version = "10.0.0"

```

警告

不不仅 `Gopkg.toml` 宣布明确的版本都 `k8s.io/apimachinery` 和 `k8s.io/api`，它也有它们覆盖。在没有从这两个存储库中显式导入包的情况下启动项目时，这是必需的。在这种情况下，没有这些覆盖 `dep` 会忽略开头的约束，开发人员会从一开始就得到错误的依赖。

即使这里显示的 `Gopkg.toml` 文件在技术上也不正确，因为它不完整，因为它没有声明对所需的所有其他库的依赖性 `client-go`。在过去，上游图书馆打破了编译 `client-go`。因此，如果您使用 `dep` 依赖项管理，请为此做好准备。

去模块

去模块是Golang中依赖管理的未来。它们在Go 1.11中得到初步支持，并进一步稳定在1.12。许多命令，如 `go run` 和 `go get`，通过设置 `GOP111MODULE=on` 环境变量来使用Go模块。在Go 1.13中，这将是默认设置。

Go模块由项目根目录中的`go.mod`文件驱动。以下是第8章中`github.com/programming-kubernetes/pizza-apiserver`项目的`go.mod`文件的摘录：

```

模块github.com/programming-kubernetes/pizza-apiserver

要求 (
  ...
  k8s.io/api v0.0.0-2019022213804-5cb15d344471 //间接
  k8s.io/apimachinery v0.0.0-20190221213512-86fb29eff628
  k8s.io/apiserver v0.0.0-20190319190228-a4358799e4fe
  k8s.io/client-go v2.0.0-alpha.0.0.20190307161346-7621a5ebb88b +不兼容
  k8s.io/klog v0.2.1-0.20190311220638-291f19f84ceb
  k8s.io/kube-openapi v0.0.0-20190320154901-c59034cc13d5 //间接
  k8s.io/utils v0.0.0-20190308190857-21c4ce38f2a7 //间接

```

```
    sigs.k8s.io/yaml v1.1.0 //间接  
)
```

`client-go v11.0.0` 匹配的Kubernetes 1.14及更早版本没有对Go模块的明确支持。仍然可以将Go模块与Kubernetes库一起使用，如前面的示例所示。

但是，只要 `client-go` 和其他Kubernetes存储库不发送`go.mod`文件（至少在Kubernetes 1.15之前），必须手动选择正确的版本。也就是说，你需要匹配的依赖的修订所有依赖的完整列表`Godeps` / `Godeps.json`在 `client-go`。

另请注意前一个示例中不太易读的修订版。它们是从现有标签派生的伪版本，或者 `v0.0.0` 如果没有标签则用作前缀。更糟糕的是，您可以在该文件中引用标记版本，但Go模块命令将替换下次运行时使用伪版本的命令。

通过 `client-go v12.0.0` 匹配Kubernetes 1.15，我们发布了一个`go.mod`文件并弃用了对所有其他销售工具的支持（参见[相应的提案文档](#)）。随附的`go.mod`文件包含所有依赖项，您的项目`go.mod`文件不再需要手动列出所有传递依赖项。在以后的版本中，也可能会更改标记方案以修复丑陋的伪修订并用适当的替换它们semver标签。但在撰写本文时，尚未完全实施或决定。

摘要

在本章中，我们的重点是Go中的Kubernetes编程接口。我们讨论了访问众所周知的核心类型Kubernetes API，即每个Kubernetes集群附带的API对象。

有了这个，我们已经介绍了Kubernetes API的基础知识及其在Go中的表示。现在，我们已准备好继续讨论自定义资源这一主题，这是运营商的支柱之一。

¹请参阅“[API术语](#)”。

² `kubectl explain pod` 允许您在API服务器中查询对象的模式，包括字段文档。

第4章使用自定义资源

在本章我们向您介绍自定义资源（CR），这是整个Kubernetes生态系统中使用的中心扩展机制之一。

自定义资源用于小型的内部配置对象，没有任何相应的控制器逻辑 - 纯粹以声明方式定义。但是，对于希望提供Kubernetes原生API体验的Kubernetes之上的许多重要开发项目，自定义资源也发挥着核心作用。示例是服务网格，例如Istio、Linkerd 2.0和AWS App Mesh，它们都有自定义资源。

还记得[第1章中的“动机例子”](#)吗？它的核心是它有一个如下所示的CR：

```
apiVersion: cnat.programming-kubernetes.info/v1alpha1
kind: At
metadata:
  name: example-at
spec:
  schedule: "2019-07-03T02:00:00Z"
status:
  phase: "pending"
```

习惯自1.7版以来，每个Kubernetes集群都提供了资源。它们存储在与 `etcd` 主Kubernetes API资源相同的实例中，并由相同的Kubernetes API服务器提供服务。[如图4-1所示](#)，请求回退到为 `apiextensions-apiserver` 资源提供服务的请求通过CRD定义，如果它们不是以下两者：

- 由聚合的API服务器处理（参见[第8章](#)）。
- 本地Kubernetes资源。

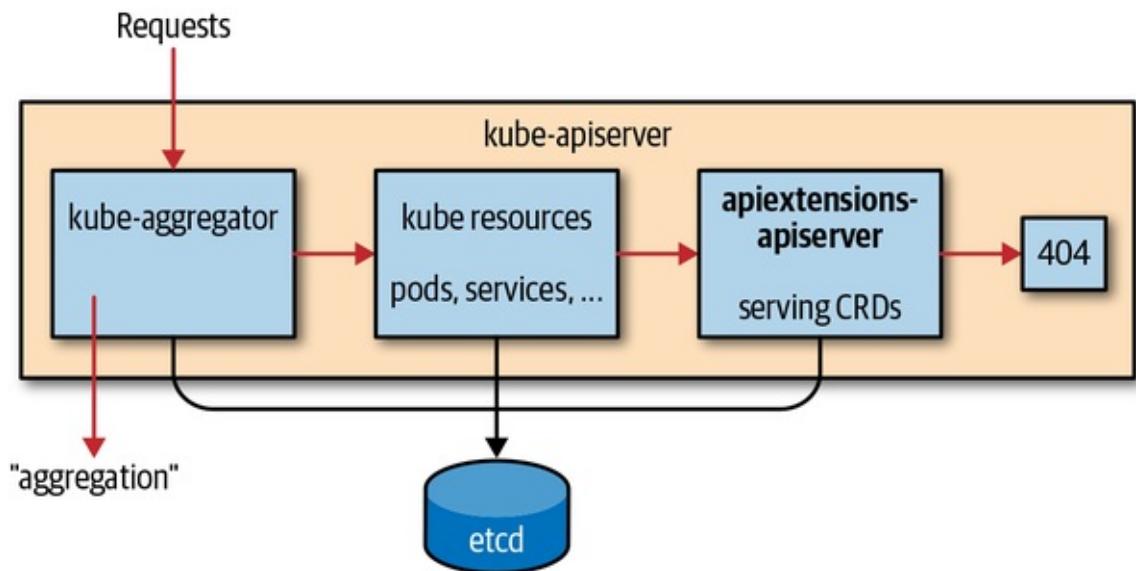


图4-1。Kubernetes API服务器内的API Extensions API服务器

CustomResourceDefinition（CRD）本身就是Kubernetes资源。它描述了群集中的可用CR。对于前面的示例CR，相应的CRD如下所示：

```

apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: ats.cnat.programming-kubernetes.info
spec:
  group: cnat.programming-kubernetes.info
  names:
    kind: At
    listKind: AtList
    plural: ats
    singular: at
  scope: Namespaced
  subresources:
    status: {}
  version: v1alpha1
  versions:
    - name: v1alpha1
      served: true
      storage: true

```

在这种情况下，CRD的名称 - `ats.cnat.programming-kubernetes.info` 必须匹配复数名称，后跟组名称。它将 `At` API组中的种类CR 定义 `cnat.programming-kubernetes.info` 为名为的命名空间资源 `ats`。

如果在群集中创建此CRD，`kubectl` 将自动检测资源，用户可以通过以下方式访问它：

```

$ kubectl得到了
姓名创建于
ats.cnat.programming-kubernetes.info 2019-04-01T14: 03: 33Z

```

发现信息

背后场景，`kubectl` 使用来自API服务器的发现信息来了解新资源。让我们看一下这个发现机制。

在增加详细级别后 `kubectl`，我们实际上可以看到它如何了解新的资源类型：

```

$ kubectl得到ats -v =7
... GET https://XXX.eks.amazonaws.com/apis/cnat.programming-kubernetes.info/
          v1alpha1 / namespaces / cnat / ats? limit =50
0
... 请求标题:
... 接受: application / json ;as=表;v=v1beta1 ;g=meta.k8s.io, application / json
        User-Agent: kubectl / v1.14.0 (darwin / amd64 )kubernetes / 641856d
... 响应状态: 200以毫秒为单位确定
姓名年龄
例子 - 在43s

```

详细的发现步骤是：

1. 最初，`kubectl` 不知道 `ats`。
2. 因此，`kubectl` 通过/`apis`发现端点向API服务器询问所有现有API组。
3. 接下来，`kubectl` 通过/`apis /group version` group发现端点向API服务器询问所有现有API组中的资源。
4. 然后，`kubectl` 将给定类型转换 `ats` 为以下三倍：
 - 集团（这里 `cnat.programming-kubernetes.info`）
 - 版本（这里 `v1alpha1`）
 - 资源（这里 `ats`）。

发现端点提供了在最后一步执行转换所需的所有信息：

```
$ http localhost: 8080 / apis /
{
  "群组": [
    {
      "name": "at.cnat.programming-kubernetes.info",
      "preferredVersion": {
        "groupVersion": "cnat.programming-kubernetes.info/v1",
        "版本": "v1alpha1"
      },
      "版本": [
        {
          "groupVersion": "cnat.programming-kubernetes.info/v1alpha1",
          "版本": "v1alpha1"
        }
      ],
      ...
    }
  ]
}

$ http localhost: 8080 / apis / cnat.programming-kubernetes.info / v1alpha1
{
  "apiVersion": "v1",
  "groupVersion": "cnat.programming-kubernetes.info/v1alpha1",
  "kind": "APIResourceList",
  "资源": [
    {
      "善良": "在",
      "名字": "ats",
      "namespaced": 是的,
      "动词": ["创建", "删除", "删除集合",
                "get", "list", "patch", "update", "watch"
              ]
    },
    ...
  ]
}
```

这一切都是由发现实现的 `RESTMapper`。我们也看到了这个非常常见的类型 `RESTMapper` 在“[REST映射](#)”。

警告

该 `kubectl` CLI 还保持资源类型的高速缓存中的 `~/.kubectl`，使其不必再检索每次访问发现信息。此缓存每10分钟无效。因此，CRD的更改可能会在相应用户的CLI中显示，最多10分钟后。

类型定义

现在让我们更详细地看一下CRD和提供的功能：如 `cnat` 示例中所示，CRD是Kubernetes API服务器进程内部 `apiextensions.k8s.io/v1beta1` 提供的API组中的Kubernetes资源 `apiextensions-apiserver`。

CRD的架构如下所示：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: name
spec:
  group: group name
  version: version name
  names:
    kind: uppercase name
    plural: lowercase plural name
    singular: lowercase singular name # defaulted to be lowercase kind
    shortNames: list of strings as short names # optional
    listKind: uppercase list kind # defaulted to be kindList
    categories: list of category membership like "all" # optional
  validation: # optional
  openAPIV3Schema: OpenAPI schema # optional
  subresources: # optional
    status: {} # to enable the status subresource (optional)
    scale: # optional
      specReplicasPath: JSON path for the replica number in the spec of the
                        custom resource
      statusReplicasPath: JSON path for the replica number in the status of
                        the custom resource
      labelSelectorPath: JSON path of the Scale.Status.Selector field in the
                        scale resource
  versions: # defaulted to the Spec.Version field
  - name: version name
    served: boolean whether the version is served by the API server # defaults to f
  else
    storage: boolean whether this version is the version used to store object
  - ...
```

许多字段是可选的或默认的。我们将在以下部分中更详细地解释这些字段。

后创建一个CRD对象，`apiextensions-apiserver` 内部 `kube-apiserver` 将检查名称并确定它们是否与其他资源冲突或者它们本身是否一致。片刻之后，它会将结果报告给CRD的状态，例如：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: ats.cnat.programming-kubernetes.info
spec:
  group: cnat.programming-kubernetes.info
  names:
    kind: At
    listKind: AtList
    plural: ats
    singular: at
  scope: Namespaced
  subresources:
    status: {}
  validation:
    openAPIV3Schema:
      type: object
      properties:
        apiVersion:
          type: string
        kind:
          type: string
        metadata:
          type: object
        spec:
          properties:
            schedule:
              type: string
            type: object
        status:
          type: object
  version: v1alpha1
  versions:
  - name: v1alpha1
    served: true
    storage: true
status:
  acceptedNames:
    kind: At
    listKind: AtList
    plural: ats
    singular: at
  conditions:
  - lastTransitionTime: "2019-03-17T09:44:21Z"
    message: no conflicts found
    reason: NoConflicts
    status: "True"
    type: NamesAccepted
  - lastTransitionTime: null
    message: the initial names have been accepted
```

```
reason: InitialNamesAccepted
status: "True"
type: Established
storedVersions:
- v1alpha1
```

您可以看到规范中缺少的名称字段是默认的，并在状态中反映为可接受的名称。此外，还设置了以下条件：

- `NamesAccepted` 描述规范中给定的名称是否一致且没有冲突。
- `Established` 描述API服务器在名称下提供给定资源 `status.acceptedNames`。

请注意，在创建CRD后很长时间内可以更改某些字段。例如，您可以添加短名称或列。在这种情况下，可以建立CRD，即使用旧名称提供 - 尽管规范名称存在冲突。因此，`NamesAccepted` 条件将是错误的，规范名称和接受的名称将不同。

自定义资源的高级功能

在本节中，我们将讨论自定义资源的高级功能，例如验证或子资源。

验证自定义资源

CR可以在创建和更新期间由API服务器进行验证。这是基于CRD规范中的字段中指定的[OpenAPI v3架构](#)完成的 `validation`。

当请求创建或改变CR时，规范中的JSON对象将根据此规范进行验证，如果出现错误，则会在HTTP代码 `400` 响应中将冲突字段返回给用户。[图4-2](#)显示了在...内的请求处理程序中进行验证的位置 `apiextensions-apiserver`。

可以在验证准入webhooks中实现更复杂的验证 - 即，使用图灵完整的编程语言。[图4-2](#)显示了在本节中描述的基于OpenAPI的验证之后直接调用这些webhook。在“[Admission Webhooks](#)”中，我们将看到如何实施和部署入场 webhook。在那里，我们将研究将其他资源考虑在内的验证，因此远远超出OpenAPI v3验证。幸运的是，对于许多用例，OpenAPI v3模式就足够了。

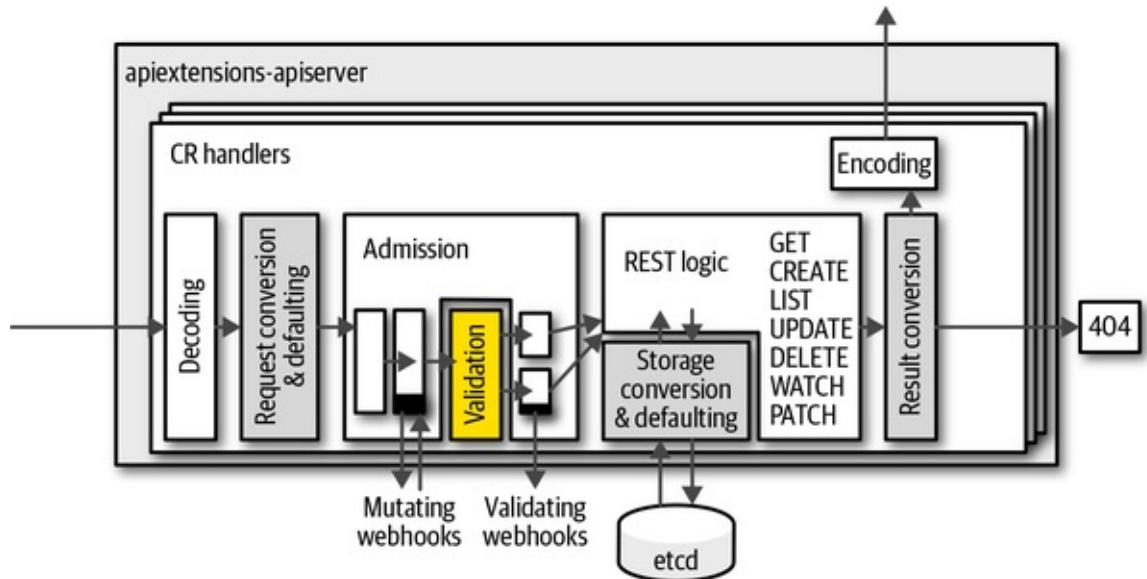


图4-2。apiextensions-apiserver的处理程序堆栈中的验证步骤

该OpenAPI模式语言基于[JSON Schema标准](#)，该[标准](#)使用JSON / YAML本身来表示模式。这是一个例子：

```

type: object
properties:
  apiVersion:
    type: string
  kind:
    type: string
  metadata:
    type: object
  spec:
    type: object
    properties:
      schedule:
        type: string
        pattern: "^\d{4}-([0]\d|1[0-2])-([0-2]\d|3[01])...""
      command:
        type: string
      required:
        - schedule
        - command
  status:
    type: object
    properties:
      phase:
        type: string
  required:
    - metadata
    - apiVersion
    - kind
    - spec
  
```

此模式指定该值实际上是JSON对象; 即, 它是一个字符串映射, 而不是一个切片或一个像数字的值。此外, 它具有 (除了 `metadata`, `kind`, 和 `apiVersion`, 其中隐式地定制资源定义的) 两个附加属性: `spec` 和 `status`。

每个都是JSON对象。`spec` 有需要的领域 `schedule` 和 `command`, 这两者都是字符串。`schedule` 必须匹配ISO日期的模式 (在这里用一些正则表达式勾画)。optional `status` 属性有一个名为的字符串字段 `phase`。

OPENAPI V3架构, 完整性及其未来

OpenAPI v3模式曾经是CRD中的可选模式。在Kubernetes 1.14之前, 它们仅用于服务器端验证。为此目的, 它们也可能是不完整的 - 换句话说, 它们可能没有指定所有字段。

从Kubernetes 1.15开始, CRD模式将作为Kubernetes API服务器OpenAPI规范的一部分发布。这个特别 `kubectl` 用于客户端验证。客户端验证会抱怨未知字段。例如, 当用户键入 `foo:bar` 对象并且 OpenAPI架构未指定时 `foo`, `kubectl` 将拒绝该对象。因此, 优良作法是传递完整的OpenAPI模式。

最后, [将来将修剪自定义资源实例](#)。这意味着 - 类似于本地Kubernetes资源类型的pod-未知 (未指定) 字段将不会被持久化。这不仅对数据一致性很重要, 而且对安全性也很重要。这是CRD的 OpenAPI模式应该完整的另一个原因。

有关完整参考, 请参阅[OpenAPI v3架构文档](#)。

手动创建OpenAPI模式可能很乏味。幸运的是, 正在进行的工作是通过代码生成使这更容易:
Kubebuilder项目 - 参见“[Kubebuilder](#)” - 已 `crd-gen` 在[sig.k8s.io/controller-tools](#)中开发, 并且这是逐步扩展的, 以便它可以在其他地方使用上下文。发电机 `crd-schema-gen` 是 `crd-gen` 这个方向的叉子。

短名称和类别

喜欢本机资源, 自定义资源可能具有长资源名称。它们在API级别上很棒, 但在CLI中输入很繁琐。CR也可以有短名称, 就像 `daemonsets` 可以查询的本机资源一样 `kubectl get ds`。这些短名称也称为别名, 每个资源可以包含任意数量的别名。

至查看所有可用的短名称, 使用如下 `kubectl api-resources` 命令:

```
$ kubectl api-resources
NAME SHORTNAMES APIGROUP NAMESPACED KIND
绑定
componentstatuses cs
configmaps cm
端点ep
事件ev
limitranges限制
名称空间ns
节点没有
true      true      true      true      true      true
false     false     false     false     false     false
绑定
ComponentStatus
ConfigMap
端点
事件
LimitRange
命名空间
节点
```

<code>persistentvolumeclaims</code>	<code>pvc</code>	<code>true</code>	<code>PersistentVolumeClaim</code>
<code>persistentvolumes</code>	<code>pv</code>	<code>false</code>	<code>PersistentVolume</code>
<code>pods</code>	<code>po</code>	<code>true</code>	<code>pod</code>
<code>statefulsets</code>	<code>sts</code>	<code>apps</code>	<code>StatefulSet</code>
...			

再次，`kubectl` 了解短名称通过发现信息（参见“[发现信息](#)”）。这是一个例子：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: ats.cnat.programming-kubernetes.info
spec:
  ...
  shortNames:
  - at
```

之后，`a kubectl get at` 将列出 `cnat` 命名空间中的所有CR。

此外，CR与任何其他资源一样 - 可以是其中的一部分类别。最常见的用途是 `all` 类别，如 `kubectl get all`。它列出了群集中所有面向用户的资源，如pod和服务。

群集中定义的CR可以通过以下 `categories` 字段加入类别或创建自己的类别：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: ats.cnat.programming-kubernetes.info
spec:
  ...
  categories:
  - all
```

有了这个，`kubectl get all` 还会 `cnat` 在命名空间中列出CR。

打印机列

该 `kubectl` CLI工具使用服务器端打印来呈现输出 `kubectl get`。这意味着它向API服务器查询要显示的列以及每行中的值。

自定义资源也支持服务器端打印机列 `additionalPrinterColumns`。它们被称为“附加”，因为第一列始终是对象的名称。这些列的定义如下：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: ats.cnat.programming-kubernetes.info
```

```

spec:
  additionalPrinterColumns: (optional)
  - name: kubectl column name
    type: OpenAPI type for the column
    format: OpenAPI format for the column (optional)
    description: human-readable description of the column (optional)
    priority: integer, always zero supported by kubectl
    JSONPath: JSON path inside the CR for the displayed value

```

该 `name` 字段是列名，`type` 是规范的[数据类型](#)部分中定义的OpenAPI类型，并且 `format`（在同一文档中定义）是可选的，可能由 `kubectl` 其他客户端解释。

此外，`description` 是一个可选的人类可读字符串，用于文档目的。显示列的 `priority` 详细模式的控件 `kubectl`。在撰写本文时（使用Kubernetes 1.14），仅支持零，并且隐藏所有具有更高优先级的列。

最后，`JSONPath` 定义要显示的值。它是CR内部的简单JSON路径。这里，“简单”意味着它支持对象字段语法 `.spec.foo.bar`，但不支持循环遍历数组或类似的更复杂的JSON路径。

有了这个，引言中的示例CRD可以 `additionalPrinterColumns` 像这样扩展：

```

additionalPrinterColumns: #(optional)
- name: schedule
  type: string
  JSONPath: .spec.schedule
- name: command
  type: string
  JSONPath: .spec.command
- name: phase
  type: string
  JSONPath: .status.phase

```

然后 `kubectl` 将呈现 `cnat` 如下资源：

```

$ kubectl得到了
名称SCHEDULER命令阶段
foo 2019-07-03T02: 00: 00Z echo "hello world" 待定

```

接下来，我们来看看子资源。

子资源

我们简要提到了[“Status Subresources: UpdateStatus”](#)中的子资源。子资源是特殊的HTTP端点，使用附加到普通资源的HTTP路径的后缀。例如，`pod`标准HTTP路径是`/api/v1/namespace/namespace/pods/name`。Pod有许多子资源，例如`/logs`, `/portforward`, `/exec`和`/status`。相应的子资源HTTP路径是：

- /api/v1/namespace/`namespace`/pods/`name`/logs
- /api/v1/namespace/`namespace`/pods/`name`/portforward
- /api/v1/namespace/`namespace`/pods/`name`/exec
- /api/v1/namespace/`namespace`/pods/`name`/status

子资源端点使用与主资源端点不同的协议。

在撰写本文时，自定义资源支持两个子资源：/scale和/status。两者都是选择性的，即必须在CRD中明确启用它们。

状态子资源

该状态子资源用于从控制器提供的状态中拆分用户提供的CR实例规范。这样做的主要动机是特权分离：

- 用户通常不应该写状态字段。
- 控制器不应写入规范字段。

该用于访问控制的RBAC机制不允许在该详细级别上的规则。这些规则始终是每个资源。该状态子资源通过提供两个端点是自己的资源解决这个问题。每个都可以独立地使用RBAC规则进行控制。这通常称为规范状态拆分。以下是ats资源的此类规则的示例，该规则仅适用于/status子资源（同时“ats”与主资源匹配）：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata: ...
rules:
- apiGroups: [""]
  resources: ["ats/status"]
  verbs: ["update", "patch"]
```

具有/status子资源的资源（包括自定义资源）已更改语义，也适用于主资源端点：

- 它们忽略在创建期间对主HTTP端点上的状态的更改（在创建期间刚刚删除状态）和更新。
- 同样，/status子资源端点忽略有效负载状态之外的更改。无法在/status端点上创建操作。
- 每当外部 metadata 和外部的 status 变化（这尤其意味着规范的变化）时，主资源端点将增加该 metadata.generation 值。这可以用作控制器的触发器，指示用户期望已经改变。

注意通常都 spec 和 status 在更新请求被发送，但在技术上你可以留出一个请求负载相应的另一个部分。

另请注意，/status端点将忽略状态之外的所有内容，包括标签或注释等元数据更改。

自定义资源的规范状态拆分启用如下：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
spec:
  subresources:
```

```
status: {}
...

```

请注意 `status`， YAML 片段中的字段被分配了空对象。这是设置没有其他属性的字段的方法。只是写作

```
subresources:
  status:
```

将导致验证错误，因为在 YAML 中，结果是 `null` 值 `status`。

警告

启用规范状态拆分是 API 的重大变化。旧控制器将写入主端点。他们不会注意到从激活分割的位置始终忽略状态。同样，在激活拆分之前，新控制器无法写入新的 /status 端点。

在 Kubernetes 1.13 及更高版本中，可以为每个版本配置子资源。这允许我们引入 /status 子资源而不会发生重大变化：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
spec:
  ...
  versions:
    - name: v1alpha1
      served: true
      storage: true
    - name: v1beta1
      served: true
      subresources:
        status: {}
```

这将启用 /status 子资源 `v1beta1`，但不能用于 `v1alpha1`。

注意

乐观并发语义（参见“[乐观并发](#)”）与主要资源端点相同；也就是说，`status` 并且 `spec` 共享相同的资源版本计数器和 /status 更新可能由于写入主资源而发生冲突，反之亦然。换句话说，存储层上没有分割 `spec` 和分割 `status`。

扩展子资源

该可用于自定义资源的第二个子资源是 /scale。的 /scale 子资源为（投影）[2](#) 上的资源视图，使我们能够查看和修改仅复制值。这个 subresource 以 Kubernetes 中的部署和副本集等资源而闻名，显然可以扩展和缩小。

该 `kubectl scale` 命令使用 /scale 子资源；例如，以下内容将修改给定实例中的指定副本值：

```
$ kubectl scale --replicas=3 your-custom-resource -v=7
I0429 21:17:53.138353 66743 round_tripppers.go:383] PUT
https://host/apis/group/v1/your-custom-resource/scale
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
spec:
  subresources:
    scale:
      specReplicasPath: .spec.replicas
      statusReplicasPath: .status.replicas
      labelSelectorPath: .status.labelSelector
...
...
```

这样，`spec.replicas` 在期间，将副本值的更新写入并从那里返回 `GET`。

标签选择器不能通过`/status`子资源更改，只能读取。其目的是为控制器提供计算相应用对象的信息。例如，`ReplicaSet` 控制器计算满足此选择器的相应pod。

标签选择器是可选的。如果您的自定义资源语义不适合标签选择器，则不要为其指定JSON路径。

在前面 `kubectl scale --replicas=3 ...` 的值示例 3 写入 `spec.replicas`。当然，可以使用任何其他简单的JSON路径；例如，`spec.instances` 或者 `spec.size` 是一个合理的字段名称，具体取决于上下文。

副本整数值与创建和删除副本的控制器

我们只讨论在自定义资源中读取和设置副本整数值。其背后的实际语义 - 例如，创建和删除实际副本的实例 - 必须由自定义控制器实现（请参阅[“控制器和操作员”](#)）。

从端点读取或写入的对象类型 `Scale` 来自 `autoscaling/v1` API组。这是它的样子：

```
type Scale struct {
    metav1.TypeMeta `json:",inline"`
    // Standard object metadata; More info: https://git.k8s.io/
    // community/contributors/devel/api-conventions.md#metadata.
    // +optional
    metav1.ObjectMeta `json:"metadata,omitempty"`

    // defines the behavior of the scale. More info: https://git.k8s.io/community/
    // contributors/devel/api-conventions.md#spec-and-status.
    // +optional
    Spec ScaleSpec `json:"spec,omitempty"`

    // current status of the scale. More info: https://git.k8s.io/community/
    // contributors/devel/api-conventions.md#spec-and-status. Read-only.
    // +optional
    Status ScaleStatus `json:"status,omitempty"`
}

// ScaleSpec describes the attributes of a scale subresource.
```

```

type ScaleSpec struct {
    // desired number of instances for the scaled object.
    // +optional
    Replicas int32 `json:"replicas,omitempty"`
}

// ScaleStatus represents the current status of a scale subresource.
type ScaleStatus struct {
    // actual number of observed instances of the scaled object.
    Replicas int32 `json:"replicas"`

    // label query over pods that should match the replicas count. This is the
    // same as the label selector but in the string format to avoid
    // introspection by clients. The string will be in the same
    // format as the query-param syntax. More info about label selectors:
    // http://kubernetes.io/docs/user-guide/labels#label-selectors.
    // +optional
    Selector string `json:"selector,omitempty"`
}

```

实例将如下所示：

```

metadata:
  name: cr-name
  namespace: cr-namespace
  uid: cr-uid
  resourceVersion: cr-resource-version
  creationTimestamp: cr-creation-timestamp
spec:
  replicas: 3
  status:
    replicas: 2
    selector: "environment = production"

```

请注意，主要资源和`/scale`子资源的乐观并发语义相同。也就是说，主要资源写入可能与`/scale`写入冲突，反之亦然。

开发人员对自定义资源的看法

自定义资源可以使用许多客户端从Golang访问。我们将专注于：

- 使用 `client-go` 动态客户端（请参阅[“动态客户端”](#)）
- 使用键入的客户端：
 - 由[kubernetes-sigs / controller-runtime](#)提供，并由Operator SDK和Kubebuilder使用（请参阅[“Operator SDK和Kubebuilder的控制器 - 运行时客户端”](#)）
 - 正如[k8s.io/client-go/kubernetes](#)中所生成的 `client-gen` 那样（参见[“通过client-gen创建的类型客户端”](#)）

选择使用哪个客户端主要取决于要编写的代码的上下文，尤其是实现的逻辑和要求的复杂性（例如，动态和支持在编译时未知的GVK）。

前面的客户列表：

- 降低处理未知GVK的灵活性。
- 增加类型安全性。
- 增加了他们提供的Kubernetes API功能的完整性。

动态客户端

该[k8s.io/client-go/dynamic](#)中的动态客户端与已知的GVK完全无关。除了[unstructured.Unstructured](#)之外，它甚至不使用任何Go类型，它包含just `json.Unmarshal` 和它的输出。

动态客户端既不使用方案也不使用RESTMapper。这意味着开发人员必须通过以GVR的形式提供资源（请参阅“[资源](#)”）来手动提供有关类型的所有知识：

```
schema.GroupVersionResource{
    Group: "apps",
    Version: "v1",
    Resource: "deployments",
}
```

如果可以使用REST客户端配置（请参阅“[创建和使用客户端](#)”），可以在一行中创建动态客户端：

```
client, err := NewForConfig(cfg)
```

对给定GVR的REST访问非常简单：

```
client.Resource(gvr).
    Namespace(namespace).Get("foo", metav1.GetOptions{})
```

这使您可以 `foo` 在给定的命名空间中进行部署。

注意

您必须知道资源的范围（即，它是命名空间还是集群作用域）。集群范围的资源只是忽略了 `Namespace(namespace)` 调用。

动态客户端的输入和输出 `*unstructured.Unstructured` 是一个对象，它包含 `json.Unmarshal` 在解组时输出的相同数据结构：

- 对象用表示 `map[string]interface{}`。
- 数组表示为 `[]interface{}`。
- 原始类型 `string`，`bool`，`float64`，或 `int64`。

该方法 `UnstructuredContent()` 提供对非结构化对象内部的数据结构的访问（我们也可以访问 `Unstructured.Object`）。在同一个包中有助手可以轻松检索字段并可以对对象进行操作 - 例如：

```
name, found, err := unstructured.NestedString(u.Object, "metadata", "name")
```

"foo" 在这种情况下返回部署的名称。 `found` 如果实际找到该字段（不仅是空的，而是实际存在的），则为真。 `err` 报告现有字段的类型是否是意外的（即，在这种情况下不是字符串）。其他助手是通用助手，一次是结果的深层副本，一次是没有：

```
func NestedFieldCopy(obj map[string]interface{}, fields ...string)
    (interface{}, bool, error)
func NestedFieldNoCopy(obj map[string]interface{}, fields ...string)
    (interface{}, bool, error)
```

还有其他类型的变体进行类型转换，如果失败则返回错误：

```
func NestedBool(obj map[string]interface{}, fields ...string) (bool, bool, error)
func NestedFloat64(obj map[string]interface{}, fields ...string)
    (float64, bool, error)
func NestedInt64(obj map[string]interface{}, fields ...string) (int64, bool, error)
func NestedStringSlice(obj map[string]interface{}, fields ...string)
    ([]string, bool, error)
func NestedSlice(obj map[string]interface{}, fields ...string)
    ([]interface{}, bool, error)
func NestedStringMap(obj map[string]interface{}, fields ...string)
    (map[string]string, bool, error)
```

最后一个通用的setter：

```
func SetNestedField(obj, value, path...)
```

动态客户端在Kubernetes中用于通用控制器，如垃圾收集控制器，它删除父项已消失的对象。垃圾收集控制器可以与系统中的任何资源一起使用，因此可以广泛使用动态客户端。

键入的客户端

键入的客户端不要使用 `map[string]interface{}` 类似的通用数据结构，而是使用真实的Golang类型，每个GVK都不同且具体。它们更易于使用，大大提高了类型安全性，并使代码更简洁，更易读。缺点是，它们的灵活性较低，因为必须在编译时知道已处理的类型，并生成这些客户端，这会增加复杂性。

在进入类型化客户端的两个实现之前，让我们看看Golang类型系统中各种类型的表示（有关Kubernetes类型系统背后的理论，请参阅[“深度API机械”](#)）。

一种类型的解剖

种表示为Golang结构。通常结构被命名为类型（虽然从技术上讲它不必是），并且被放置在与手头的GVK的组和版本相对应的包中。常见的惯例是放置GVK *group/ version*。Kind进入Go包：

```
pkg / apis / group / version
```

并Kind在文件types.go中定义Golang结构。

对应于GVK的每个Golang类型都嵌入 TypeMeta 了包[k8s.io/apimachinery/pkg/apis/meta/v1](#)中的结构。TypeMeta 只包括 Kind 和 ApiVersion 字段：

```
type TypeMeta struct {
    // +optional
    APIVersion string `json:"apiVersion,omitempty" yaml:"apiVersion,omitempty"`
    // +optional
    Kind string `json:"kind,omitempty" yaml:"kind,omitempty"`
}
```

此外，每个顶级类型 - 即具有自己的端点并因此具有一个（或多个）对应的GVR（参见“[REST映射](#)”） - 具有存储名称，命名空间资源的命名空间和漂亮的类型。大量进一步的元级域。所有这些都存储在[k8s.io/apimachinery/pkg/apis/meta/v1](#) ObjectMeta 包中调用的结构中：

```
type ObjectMeta struct {
    Name string `json:"name,omitempty"`
    Namespace string `json:"namespace,omitempty"`
    UID types.UID `json:"uid,omitempty"`
    ResourceVersion string `json:"resourceVersion,omitempty"`
    CreationTimestamp Time `json:"creationTimestamp,omitempty"`
    DeletionTimestamp *Time `json:"deletionTimestamp,omitempty"`
    Labels map[string]string `json:"labels,omitempty"`
    Annotations map[string]string `json:"annotations,omitempty"`
    ...
}
```

还有许多其他字段。我们强烈建议您阅读[大量的内联文档](#)，因为它可以很好地描述Kubernetes对象的核心功能。

Kubernetes顶级类型（即那些具有嵌入式 TypeMeta 和嵌入式 ObjectMeta，并且在这种情况下持久化的类型 etcd）看起来非常相似，因为它们通常具有a spec 和a status。从[k8s.io/kubernetes/apps/v1/types.go](#)查看此部署示例：

```
type Deployment struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec DeploymentSpec `json:"spec,omitempty"`
    Status DeploymentStatus `json:"status,omitempty"`
}
```

虽然不同类型的类型的实际内容 spec 和 status 不同类型之间存在显着差异，但这在Kubernetes中分为 spec 并且 status 是一个共同主题甚至是惯例，尽管它在技术上并不需要。因此，优良作法也是遵循这种CRD结构。一些CRD功能甚至需要这种结构；例如，自定义资源的 /status 子资源（请参阅“[状态子资源](#)”） - 启用时 - 始终仅应用于 status 自定义资源实例的子结构。它无法重命名。

GOLANG封装结构

如我们已经看到，Golang类型传统上放在包 `pkg/apis/` 中名为 `types.go` 的文件中。除了该文件之外，还有一些我们想要浏览的文件。其中一些是由开发人员手动编写的，而另一些是使用代码生成器生成的。详细信息请参见[第5章。groupversion](#)

该 `doc.go` 文件描述了 API 的目的，包括许多包全局代码生成标签：

```
// Package v1alpha1 contains the cnat v1alpha1 API group
//
// +k8s:deepcopy-gen=package
// +groupName=cnat.programming-kubernetes.info
package v1alpha1
```

接下来，`register.go` 包含帮助程序以将自定义资源 Golang 类型注册到方案中（请参阅[“方案”](#)）：

```
package version

import (
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/runtime"
    "k8s.io/apimachinery/pkg/runtime/schema"

    group "repo/pkg/apis/group"
)

// SchemeGroupVersion is group version used to register these objects
var SchemeGroupVersion = schema.GroupVersion{
    Group: group.GroupName,
    Version: "version",
}

// Kind takes an unqualified kind and returns back a Group qualified GroupKind
func Kind(kind string) schema.GroupKind {
    return SchemeGroupVersion.WithKind(kind).GroupKind()
}

// Resource takes an unqualified resource and returns a Group
// qualified GroupResource
func Resource(resource string) schema.GroupResource {
    return SchemeGroupVersion.WithResource(resource).GroupResource()
}
```

```

var (
    SchemeBuilder = runtime.NewSchemeBuilder(addKnownTypes)
    AddToScheme   = SchemeBuilder.AddToScheme
)

// Adds the list of known types to Scheme.
func addKnownTypes(scheme *runtime.Scheme) error {
    scheme.AddKnownTypes(SchemeGroupVersion,
        &SomeKind{},
        &SomeKindList{},
    )
    metav1.AddToGroupVersion(scheme, SchemeGroupVersion)
    return nil
}

```

然后，`zz_generated.deepcopy.go` 定义在自定义资源 Golang 顶级类型深复制方法（即，`SomeKind` 与 `SomeKindList` 在前面的示例代码）。此外，所有 `substructs`（像那些为 `spec` 和 `status`）成为深可复制为好。

因为本例使用的标签 `+k8s:deepcopy-gen=package` 在 `doc.go`，深拷贝世代是选择退出的基础上；也就是说，`DeepCopy` 为包中没有选择退出的每种类型生成方法 `+k8s:deepcopy-gen=false`。有关详细信息，请参阅[第5章](#)，尤其是“[deepcopy-gen标签](#)”。

通过CLIENT-GEN创建的类型客户端

与 API 包 `PKG` 的 `/apis/group/version` 到位，客户端发生器 `client-gen` 创建一个输入客户端（参见[第5章](#)的详细信息，特别是“[客户端根标签](#)”），在 `PKG` / 生成 / `clientset` / 版本默认 (`PKG` / 客户端 / 客户端 / 版本化的旧版本的生成器)。更确切地说，生成的顶级对象是客户端集。它包含许多 API 组，版本和资源。

该[顶层文件](#)如下所示：

```

// Code generated by client-gen. DO NOT EDIT.

package versioned

import (
    discovery "k8s.io/client-go/discovery"
    rest "k8s.io/client-go/rest"
    flowcontrol "k8s.io/client-go/util/flowcontrol"

    cnatv1alpha1 ".../cnat/cnat-client-go/pkg/generated/clientset/versioned/"
)

type Interface interface {
    Discovery() discovery.DiscoveryInterface
    CnatV1alpha1() cnatv1alpha1.CnatV1alpha1Interface
}

```

```

}

// Clientset contains the clients for groups. Each group has exactly one
// version included in a Clientset.
type Clientset struct {
    *discovery.DiscoveryClient
    cnatV1alpha1 *cnatv1alpha1.CnatV1alpha1Client
}

// CnatV1alpha1 retrieves the CnatV1alpha1Client
func (c *Clientset) CnatV1alpha1() cnatv1alpha1.CnatV1alpha1Interface {
    return c.cnatV1alpha1
}

// Discovery retrieves the DiscoveryClient
func (c *Clientset) Discovery() discovery.DiscoveryInterface {
    ...
}

// NewForConfig creates a new Clientset for the given config.
func NewForConfig(c *rest.Config) (*Clientset, error) {
    ...
}

```

客户端集由接口表示，`Interface` 并为每个版本提供对 API 组客户端接口的访问权限 - 例如，`CnatV1alpha1Interface` 在此示例代码中：

```

type CnatV1alpha1Interface interface {
    RESTClient() rest.Interface
    AtsGetter
}

// AtsGetter has a method to return a AtInterface.
// A group's client should implement this interface.
type AtsGetter interface {
    Ats(namespace string) AtInterface
}

// AtInterface has methods to work with At resources.
type AtInterface interface {
    Create(*v1alpha1.At) (*v1alpha1.At, error)
    Update(*v1alpha1.At) (*v1alpha1.At, error)
    UpdateStatus(*v1alpha1.At) (*v1alpha1.At, error)
    Delete(name string, options *v1.DeleteOptions) error
    DeleteCollection(options *v1.DeleteOptions, listOptions v1.ListOptions) error
    Get(name string, options v1.GetOptions) (*v1alpha1.At, error)
    List(opts v1.ListOptions) (*v1alpha1.AtList, error)
    Watch(opts v1.ListOptions) (watch.Interface, error)
    Patch(name string, pt types.PatchType, data []byte, subresources ...string)
        (result *v1alpha1.At, err error)
}

```

```
    AtExpansion
}
```

可以使用 `NewForConfig` 辅助函数创建客户端集的实例。这类似于“[创建和使用客户端](#)”中讨论的核心 Kubernetes 资源的客户端：

```
import (
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/tools/clientcmd"

    client "github.com/.../cnat/cnat-client-go/pkg/generated/clientset/versioned"
)

kubeconfig = flag.String("kubeconfig", "~/.kube/config", "kubeconfig file")
flag.Parse()
config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
clientset, err := client.NewForConfig(config)

ats := clientset.CnatsV1alpha1Interface().Ats("default")
book, err := ats.Get("kubernetes-programming", metav1.GetOptions{})
```

如您所见，代码生成机制允许我们以与核心 Kubernetes 资源相同的方式为自定义资源编写逻辑。也可以使用像线人这样的高级工具；看 `informer-gen` 在[第5章](#)。

operator SDK 和 Kubebuilder 的 controller-runtime 客户端

对于为了完整起见，我们想快速浏览一下第三个客户端，它被列为“[开发人员对自定义资源的视图](#)”中的第二个选项。该 controller-runtime 项目为[第6章](#)中介绍的运营商解决方案 Operator SDK 和 Kubebuilder 提供了基础。它包括一个使用“[类型剖析](#)”中提供的 Go 类型的客户端。

与 `client-gen` 先前“[通过客户端创建的类型化客户端](#)”的生成客户端相比，并且类似于“[动态客户端](#)”，该客户端是一个实例，能够处理在给定方案中注册的任何类型。

它使用来自 API 服务器的发现信息将种类映射到 HTTP 路径。请注意，[第6章](#) 将详细介绍如何将此客户端用作这两个运算符解决方案的一部分。

以下是如何使用的快速示例 controller-runtime：

```
import (
    "flag"

    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes/scheme"
    "k8s.io/client-go/tools/clientcmd"

    runtimeclient "sigs.k8s.io/controller-runtime/pkg/client"
)
```

```

kubeconfig = flag.String("kubeconfig", "~/.kube/config", "kubeconfig file path")
flag.Parse()
config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)

cl, _ := runtimeclient.New(config, client.Options{
    Scheme: scheme.Scheme,
})
podList := &corev1.PodList{}
err := cl.List(context.TODO(), client.InNamespace("default"), podList)

```

客户端对象的 `List()` 方法接受 `runtime.Object` 在给定方案中注册的任何方法，在这种情况下是从 `client-go` 所有标准Kubernetes种类中注册的方案。在内部，客户端使用给定的方案将Golang类型映射 `*corev1.PodList` 到GVK。在第二步中，该 `List()` 方法使用发现信息来获取pod的GVR `schema.GroupVersionResource{ "", "v1", "pods" }`，因此访问 `/api/v1/namespace/default/pods` 以获取传递的命名空间中的pod列表。

相同的逻辑可以与自定义资源一起使用。主要区别是使用包含传递的Go类型的自定义方案：

```

import (
    "flag"

    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes/scheme"
    "k8s.io/client-go/tools/clientcmd"

    runtimeclient "sigs.k8s.io/controller-runtime/pkg/client"
    cnatv1alpha1 "github.com/.../cnat/cnat-kubebuilder/pkg/apis/cnat/v1alpha1"
)

kubeconfig = flag.String("kubeconfig", "~/.kube/config", "kubeconfig file")
flag.Parse()
config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)

crScheme := runtime.NewScheme()
cnatv1alpha1.AddToScheme(crScheme)

cl, _ := runtimeclient.New(config, client.Options{
    Scheme: crScheme,
})
list := &cnatv1alpha1.AtList{}
err := cl.List(context.TODO(), client.InNamespace("default"), list)

```

请注意 `List()` 命令的调用根本不会发生变化。

想象一下，您编写了一个使用此客户端访问许多不同类型的运算符。使用“[通过client-gen创建的类型化客户端](#)”的[类型客户端](#)，您必须将许多不同的客户端传递给操作员，使得管道代码非常复杂。相比之下，`controller-runtime` 这里介绍的客户只是所有类型的一个对象，假设它们都在一个方案中。

所有三种类型的客户端都有其用途，根据使用它们的上下文有利有弊。在处理未知对象的通用控制器中，只能使用动态客户端。在类型安全有助于强制执行代码正确性的控制器中，生成的客户端非常适合。Kubernetes项目本身有很多贡献者，即使代码的稳定性非常重要，即使它被很多人扩展和重写。如果方便和高速度和最小的管道是重要的，`controller-runtime` 客户是一个很好的选择。

摘要

我们向您介绍了自定义资源，本章中Kubernetes生态系统中使用的中心扩展机制。到目前为止，您应该很好地了解它们的功能和限制以及可用的客户端。

现在让我们继续使用代码生成来管理所述资源。

[1](#)不要在这里混淆Kubernetes和JSON对象。后者只是字符串映射的另一个术语，用于JSON和OpenAPI的上下文中。

[2](#)“投影”在这里意味着 `scale` 对象是主要资源的投影，因为它只显示某些字段并隐藏其他所有字段。

第5章自动生成代码

在本章中，您将学习如何在Go项目中使用Kubernetes代码生成器以自然的方式编写自定义资源。代码生成器在本地Kubernetes资源的实现中经常使用，我们将在这里使用相同的生成器。

为什么代码生成

Go是一种简单的语言设计。它缺乏更高级甚至类元编程的机制，以通用（即，类型无关）的方式表达不同数据类型的算法。“Go way”是使用外部代码生成。

在Kubernetes开发过程的早期阶段，随着更多资源被添加到系统中，必须重写越来越多的代码。代码生成使得维护此代码变得更加容易。很早就开始了[Gengo库](#)，最终，基于[Gengo, k8s.io / code-generator](#)被开发为外部可用的发生器集合。我们将在以下部分中使用这些生成器进行CR。

召唤发电机

通常，代码生成器在每个控制器项目中以大致相同的方式调用。只有包，组名和API版本不同。调用脚本[k8s.io/code-generator/generate-groups.sh](#)或像[hack / update-codegen.sh](#)这样的bash脚本是从构建系统向CR Go类型添加代码生成的最简单方法（参见[本书的GitHub存储库](#)）。

请注意，由于非常特殊的要求和历史原因，某些项目直接调用生成器二进制文件。对于为CR构建控制器的用例，从[k8s.io/code-generator](#)存储库调用[generate-groups.sh](#)脚本要容易得多：

```
$ vendor / k8s.io / code-generator / generate-groups.sh all \
github.com/programming-kubernetes/cnat/cnat-client-go/pkg/generated \
github.com/programming-kubernetes/cnat/cnat-client-go/pkg/apis \
cnat: v1alpha1 \
--output-base "${GOPATH}/src" \
--go-header-file "hack/boilerplate.go.txt"
```

在这里，`all` 打电话的方式 CR的所有四个标准代码生成器：

- `deepcopy-gen`

生成 `func (t *T) DeepCopy()` `*T` 和 `func (t *T) DeepCopyInto(*T)` 方法。

- `client-gen`

创建类型化的客户端集。

- `informer-gen`

为CR创建提供者，提供基于事件的界面以响应服务器上CR的更改。

- `lister-gen`

为CR提供lister，为其提供只读缓存层 GET 和 LIST 请求。

最后两个是构建控制器的基础（参见“[控制器和操作员](#)”）。这四个代码生成器构成了使用Kubernetes上游控制器使用的相同机制和包构建功能齐全的生产就绪控制器的强大基础。

注意

[k8s.io/code-generator](#)中还有一些生成器，主要用于其他上下文。例如，如果您构建自己的聚合API服务器（请参阅[第8章](#)），除了版本化类型之外，您还将使用内部类型，并且必须定义默认函数。然后，您可以通过从[k8s.io/code-generator](#)调用[generate-internal-groups.sh](#)脚本来访问这两个生成器，它们将变得相关：

- `conversion-gen`

创建用于转换的函数 内部和外部类型。

- `defaulter-gen`

处理违约某些领域的问题。

现在让我们详细查看参数 `generate-groups.sh`：

- 第二个参数是生成的客户端，列表和告密者的目标包名称。
- 第三个参数是API组的基础包。
- 第四个参数是以空格分隔的API组列表及其版本。
- `--output-base` 作为标志传递给所有生成器，以定义找到给定包的基本目录。
- `--go-header-file` 使我们能够将版权标题放入生成的代码中。

某些生成器，例如 `deepcopy-gen`，直接在API组包内创建文件。这些文件遵循标准命名方案和 `zz_generated`。前缀使得很容易将它们从版本控制系统中排除（例如，通过`.gitignore`文件），尽管大多数项目决定检查生成的文件，因为围绕代码生成器的Go工具没有很好地开发。¹

如果项目遵循[k8s.io/sample-controller](#)的模式- 这 `sample-controller` 是一个蓝图项目，复制由 Kubernetes本身内置的许多控制器建立的模式 - 那么代码生成开始于：

```
$ 黑客/ update-codegen.sh
```

将 `cnat` 在本例 `sample-controller+client-go` 中的变体“[下面的示例控制器](#)”去这条路线。

小费

通常，除了 `hack/update-codegen.sh` 脚本之外，还有一个名为的第二个脚本 `hack/verify-codegen.sh`。

此脚本调用 `hack/update-codegen.sh` 脚本并检查是否有任何更改，如果任何生成的文件不是最新的，则它将以非零返回码终止。

这在持续集成（CI）脚本中非常有用：如果开发人员意外修改了文件或者文件刚刚过时，CI会注意到并抱怨。

使用标记控制生成器

而一些代码生成器行为是通过前面描述的命令行标志控制的（特别是要处理的包），通过Go文件中的标签控制更多属性。标签是一种特殊格式的Go评论，格式如下：

```
// +some-tag
// +some-other-tag=value
```

有两种标签：

- `package` 名为`doc.go`的文件中位于行上方的全局标记
- 类型声明上方的本地标记（例如，在结构定义之上）

根据相关标签，评论的位置可能很重要。

准确地关注示例（包括注释块）

有许多标记必须位于类型（或全局标记的包行）上方的注释中，而其他标记必须与类型（或包行）分开，并且它们之间至少有一个空行。例如：

```
// +second-comment-block-tag

// +first-comment-block-tag
type Foo struct {
}
```

这种区别的原因是历史性的：Kubernetes中的API文档生成器用于不了解代码生成标记，而是仅导出第一个注释块。因此，该块中的标记将出现在API HTML文档中。

代码生成器标记解析逻辑并不总是非常一致，并且错误处理通常远非完美。虽然每个版本都有所改进，但要准备好非常精确地遵循现有示例 - 例如，空行可能很重要。

全局标签

全局标签被写入包的`doc.go`。典型的`pkg/apis/group/version/doc.go`文件如下所示：

```
// +k8s:deepcopy-gen=package

// Package v1 is the v1alpha1 version of the API.
// +groupName=cnat.programming-kubernetes.info
package v1alpha1
```

该文件的第一行告诉`deepcopy-gen`默认情况下为该包中的每个类型创建深层复制方法。如果您的类型不需要深层复制，不需要，甚至不可能，您可以使用本地标签选择使用深层复制`// +k8s:deepcopy-gen=false`。如果您不启用软件包范围的深层复制，则必须通过以下方式为每个所需类型选择深层复制`// +k8s:deepcopy-gen=true`。

第二个标记`// +groupName=example.com`定义了完全限定的API组名称。如果Go父包名称与组名称不匹配，则此标记是必需的。

此处显示的文件实际上来自 [cnat client-go 示例](#)`pkg/apis/cnat/v1alpha1/doc.go` 文件（请参阅“[以下示例控制器](#)”）。那里 `cnat` 是父包，但是 `cnat.programming-kubernetes.info` 是组名。

使用 `// +groupName` 标记，客户端生成器（请参阅[“通过client-gen创建的类型化客户端”](#)）将使用正确的HTTP路径`/apis/foo.project.example.com`生成客户端。除此之外，`+groupName` 还有 `+groupGoName` 一个定义要使用的自定义Go标识符（用于变量和类型名称）而不是父包名称。例如，默认情况下，生成器将使用大写父包名称进行标识，在我们的示例中为 `Cnat`。一个更好的标识符将是 `CNAT` “Cloud Native At。” `// +groupGoName=CNAT` 我们可以使用它而不是 `Cnat`（虽然我们在这个例子中没有这样做 - 我们一直待在那里 `Cnat`），以及该 `client-gen` 结果将如下所示：

```
type Interface interface {
    Discovery() discovery.DiscoveryInterface
    CNatV1() atv1alpha1.CNatV1alpha1Interface
}
```

本地标签

本地标签直接写在API类型之上或其上方的第二个注释块中。以下是该[示例](#)的`types.go`文件中的主要类型：`cnat`

```
// AtSpec defines the desired state of At
type AtSpec struct {
    // Schedule is the desired time the command is supposed to be executed.
    // Note: the format used here is UTC time https://www.utctime.net
    Schedule string `json:"schedule,omitempty"`
    // Command is the desired command (executed in a Bash shell) to be executed.
    Command string `json:"command,omitempty"`
    // Important: Run "make" to regenerate code after modifying this file
}

// AtStatus defines the observed state of At
type AtStatus struct {
    // Phase represents the state of the schedule: until the command is executed
    // it is PENDING, afterwards it is DONE.
    Phase string `json:"phase,omitempty"`
    // Important: Run "make" to regenerate code after modifying this file
}

// +genclient
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// At runs a command at a given schedule.
type At struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec AtSpec `json:"spec,omitempty"`
}
```

```

    Status AtStatus `json:"status,omitempty"`
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// AtList contains a list of At
type AtList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata,omitempty"`
    Items          []At `json:"items"`
}

```

在以下部分中，我们将介绍此示例的标记。

小费

在此示例中，API文档位于第一个注释块中，而我们将标记放入第二个注释块中。如果您使用某种工具为此目的提取Go文档注释，这有助于将标记保留在API文档之外。

deepcopy-gen标签

深度复制方法生成通常默认情况下通过全局 `// +k8s:deepcopy-gen=package` 标记为所有类型启用（请参阅[“全局标记”](#)） - 也就是说，可能选择退出。但是，在前面的示例文件（实际上是整个包）中，所有API类型都需要深层复制方法。因此，我们不必在本地选择退出。

如果我们在API类型包中有一个帮助器结构（通常不鼓励保持API包清洁），我们必须禁用深度复制生成。例如：

```

// +k8s:deepcopy-gen=false
//
// Helper is a helper struct, not an API type.
type Helper struct {
    ...
}

```

runtime.Object和DeepCopyObject

那里 是一个特殊的深拷贝标记，需要更多解释：

```
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
```

在[“Kubernetes Objects in Go”](#)中，我们看到 `runtime.Object`s 必须实现该 `DeepCopyObject()` `runtime.Object` 方法。原因是Kubernetes中的通用代码必须能够创建对象的深层副本。这种方法允许这样做。

历史背景

在1.8之前，该方案（参见“[方案](#)”）也保留了对特定于类型的深拷贝函数的引用，并且它具有基于反射的深拷贝实现。这两种机制都是导致许多重要且难以发现的错误的原因。因此，Kubernetes使用界面中的 `DeepCopyObject` 方法切换到静态深层复制 `runtime.Object`。

该 `DeepCopyObject()` 方法除了调用生成的 `DeepCopy` 方法之外什么都不做。后者的签名因类型而异 (`DeepCopy() *T 取决于 T`)。前者的签名总是 `DeepCopyObject() runtime.Object`：

```
func (in *T) DeepCopyObject() runtime.Object {
    if c := in.DeepCopy(); c != nil {
        return c
    } else {
        return nil
    }
}
```

将本地标记放在顶级API类型上方以生成此方法。这告诉创建这样一个方法，调用。`//+k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object``deepcopy-gen``deepcopy-gen``runtime.Object``DeepCopyObject()`

小费

在前面的例子中，两个 `At` 和 `AtList` 因为它们被用作是顶级类型 `runtime.Object` 秒。

根据经验，顶级类型是 `metav1.TypeMeta` 嵌入的类型。

碰巧其他接口需要一种深度复制的方式。例如，如果API类型具有接口类型字段，则通常会出现这种情况 `Foo`：

```
type SomeAPIType struct {
    Foo Foo `json:"foo"`
}
```

正如我们所看到的，API类型必须是可深度复制的，因此该字段也 `Foo` 必须进行深度复制。你怎么能这样做，在一个通用的方法（无类型强制转换），无添加 `DeepCopyFoo()` `Foo` 的 `Foo` 接口？

```
type Foo interface {
    ...
    DeepCopyFoo() Foo
}
```

在这种情况下，可以使用相同的标签：

```
// +k8s:deepcopy-gen:interfaces=<package>.Foo
type FooImplementation struct {
    ...
}
```

那里 `runtime.Object` 是实际使用此标记的Kubernetes源中的一些示例：

```
// +k8s:deepcopy-gen:interfaces=.../pkg/registry/rbac/reconciliation.RuleOwner
// +k8s:deepcopy-gen:interfaces=.../pkg/registry/rbac/reconciliation.RoleBinding
```

客户端标签

最后，那里是一些要控制的标签 `client-gen`，其中一个我们在前面的例子中看到 `At` 和 `AtList`：

```
// +genclient
```

它告诉 `client-gen` 为这种类型创建一个客户端（这总是选择加入）。请注意，您不必并且实际上不得将其置于 `List API` 对象的类型之上。

在我们的 `cnat` 示例中，我们使用 `/status` 子资源并使用 `updateStatus` 客户端的方法更新 CR 的状态（请参阅“[状态子资源](#)”）。存在没有状态或没有规范状态拆分的 CR 的实例。在这些情况下，以下标记可以避免生成该 `updateStatus()` 方法：

```
// +genclient:noStatus
```

警告

没有这个标签，`client-gen` 就会盲目地生成 `updateStatus()` 方法。但是，重要的是要理解，只有在 `CustomResourceDefinition` 清单中实际启用了 `/status` 子资源时，`spec-status` 拆分才有效（请参阅“[子资源](#)”）。

在客户端中单独存在该方法没有任何效果。没有更改清单的请求甚至会失败。

客户端生成器必须选择正确的 HTTP 路径，可以使用或不使用命名空间。对于群集范围的资源，您必须使用标记：

```
// +genclient:nonNamespaced
```

默认是生成命名空间客户端。同样，这必须与 CRD 清单中的范围设置相匹配。对于特殊用途客户端，您可能还希望详细控制提供的 HTTP 方法。您可以使用几个标记来执行此操作，例如：

```
// +genclient:noVerbs
// +genclient:onlyVerbs=create,delete
// +genclient:skipVerbs=get,list,create,update,patch,delete,watch
// +genclient:method=Create,verb=create,
// result=k8s.io/apimachinery/pkg/apis/meta/v1.Status
```

前三个应该是不言自明的，但最后一个需要一些解释。

上面写的这个标签的类型将是仅创建的，不会返回API类型本身，而是a `metav1.Status`。对于CR而言，这没有多大意义，但对于用Go编写的用户提供的API服务器（参见[第8章](#)），这些资源可以存在，并且它们在实践中存在。

// +genclient:method= 标记的一个常见情况是添加了一种扩展资源的方法。在“[Scale subresource](#)”中，我们描述了如何为CR启用/ `scale`子资源。以下标记创建相应的客户端方法：

```
// +genclient:method=GetScale,verb=get,subresource=scale, \
//   result=k8s.io/api/autoscaling/v1.Scale
// +genclient:method=UpdateScale,verb=update,subresource=scale, \
//   input=k8s.io/api/autoscaling/v1.Scale,result=k8s.io/api/autoscaling/v1.Scale
```

第一个标签创建了getter `GetScale`。第二个创建了setter `UpdateScale`。

注意

所有CR / `scale`子资源都 `Scale` 从`autoscaling / v1`组接收和输出类型。在Kubernetes API中，由于历史原因，有些资源使用其他类型。

informer-gen和**lister-gen**

都 `informer-gen` 并 `lister-gen` 处理 // +genclient 标签 `client-gen`。没有其他配置。选择客户端生成的每种类型都会自动获取与客户端匹配的`informer`和`listers`（如果通过`k8s.io/code-generator/generate-groups.sh`脚本调用整个生成器套件）。

Kubernetes发电机的文档有很大的改进空间，随着时间的推移肯定会慢慢完善。有关不同生成器的更多信息，查看Kubernetes本身的示例通常很有帮助 - 例如，[k8s.io / api](#)和[OpenShift API类型](#)。这两个存储库都有许多高级用例。

此外，请毫不犹豫地查看发电机本身。`deepcopy-gen` 在[main.go](#)文件中有一些文档可用。`client-gen` 在[Kubernetes贡献者文档](#)中提供了一些[文档](#)。`informer-gen` 而 `lister-gen` 目前还没有进一步的文件，但[generate-groups.sh](#)显示[每个如何调用](#)。

摘要

在本章中，我们向您展示了如何将Kubernetes代码生成器用于CR。有了这一点，我们现在转向更高级别的抽象工具 - 即编写自定义控制器和运算符的解决方案，使您能够专注于业务逻辑。

¹ Go工具在需要时不会自动运行生成，并且缺乏定义源文件和生成文件之间依赖关系的方法。

第6章编写运算符的解决方案

所以到目前为止，我们已经在“[控制器和操作员](#)”的概念层面上看过自定义控制器和运算符，在[第5章中](#)，我们将讨论如何使用Kubernetes代码生成器 - 一种处理该主题的低级方法。在本章中，我们将介绍三个解决方案 详细编写自定义控制器和操作员，并讨论更多替代方案。

使用本章中讨论的解决方案之一应该可以帮助您避免编写大量重复代码，并使您能够专注于业务逻辑，而不是样板代码。它应该让您更快地开始并提高您的工作效率。

注意

一般而言，运营商以及我们在本章中具体讨论的工具在2019年中期仍在迅速发展。虽然我们尽力而为，但您在此处看到的某些命令和/或输出可能会发生变化。考虑到这一点，并确保始终使用相应工具的最新版本，密切关注相应的问题跟踪器，邮件列表和Slack通道。

虽然有在线资源可以[比较](#)我们在此讨论的解决方案，但我们不会向您推荐具体的解决方案。但是，我们鼓励您自己评估和比较它们，并选择最适合您的组织和环境的那个。

制备

我们将使用 `cnat` (at 我们在[“动机示例”中](#)介绍的cloud-native) 作为本章中不同解决方案的运行示例。如果你想跟随，请注意我们假设你：

1. 安装了Go版本1.12或更高版本并正确设置。
2. 有权访问Kubernetes集群中的1.12或以上，或者通过在当地如版本 `kind` 或 `k3d`，或者远程通过您最喜欢的云服务提供商和 `kubectl` 配置进行访问。
3. `git clone` 我们的[GitHub存储库](#)。此处提供了完整，有效的源代码和以下部分中显示的必要命令。请注意，我们在这里展示的是如何从头开始工作。如果您想查看结果而不是自己执行这些步骤，也欢迎您克隆存储库并仅运行命令来安装CRD，安装CR并启动自定义控制器。

将这些内务管理项目排除在外，让我们跳到编写操作员：我们将 `sample-controller` 在本章中介绍，Kubebuilder和Operator SDK。

准备？让我们来吧！

以下样本控制器

让我们从 `cnav` 基于[k8s.io/sample-controller](#)的实现开始，它直接使用 `client-go` 库。在使用[k8s.io/code-generator](#)生成一个类型的客户端，告密者，一线明星，和深拷贝功能。每当自定义控制器中的API类型发生变化时 - 例如，在自定义资源中添加新字段 - 您必须使用[update-codegen.sh](#)脚本（另请参阅GitHub中的[源代码](#)）来重新生成上述源文件。`sample-controller`

警告

您可能已经注意到 `k8s.io` 被用作整本书中的基本 URL。我们在[第3章](#)介绍了它的用法；作为提醒，它实际上是 `kubernetes.io` 的别名，在 Go 包管理的上下文中，它解析为 `github.com/kubernetes`。请注意，`k8s.io` 没有自动重定向。所以，例如，`k8s.io / sample-controller` 真的意味着你应该看看 github.com/kubernetes/sample-controller，等等。

好的，让我们按照以下方式 `cnat` 使用我们的运算符。（请参阅[我们的仓库中的相应目录](#)。） `client-go`sample-controller`

引导

至开始，做一个 `go get k8s.io/sample-controller` 将源和依赖项放到你的系统上，它应该在 `$GOPATH/src/k8s.io/sample-controller` 中。

如果从头开始，将 `sample-controller` 目录的内容复制到您选择的目录中（例如，我们在 `repo` 中使用 `cnav-client-go`），您可以运行以下命令序列来构建和运行基本控制器（使用默认实现，而不是 `cnav` 业务逻辑）：

```
# build custom controller binary:  
$ 去构建-o cnav-controller。  
  
# launch custom controller locally:  
$ ./cnav-controller -kubeconfig=$HOME/.kube/config
```

此命令将启动自定义控制器并等待您注册 CRD 并创建自定义资源。我们现在就做，看看会发生什么。在第二个终端会话，输入：

```
$ kubectl apply -f artifacts / examples / crd.yaml
```

使确保 CRD 正确注册并可以这样使用：

```
$ kubectl得到crds  
姓名创建于  
foos.samplecontroller.k8s.io 2019-05-29T12: 16: 57Z
```

请注意，您可能会在此处看到其他 CRD，具体取决于您使用的 Kubernetes 发行版；但是，至少应该列出 `foos.samplecontroller.k8s.io`。

接下来，我们创建示例自定义资源 `foo.samplecontroller.k8s.io/example-foo` 并检查控制器是否完成其工作：

```
$ kubectl apply -f artifacts / examples / example-foo.yaml  
创建了foo.samplecontroller.k8s.io/example-foo  
  
$ kubectl得到po, rs, deploy, foo  
NAME READY STATUS RESTARTS AGE  
pod / example-foo-5b8c9679d8-xjhdf 1/1运行 0 67s
```

```

NAME希望当前的准备好年龄
replicaset.extensions / example-foo-5b8c9679d8      1          1          1      67s

名称准备好最新可用年龄
deployment.extensions / example-foo 1/1          1          1      67s

姓名年龄
foo.samplecontroller.k8s.io/example-foo 67s

```

是的，它按预期工作！我们现在可以继续实现实际的 `cnat` 特定业务逻辑。

商业逻辑

至开始实现业务逻辑，我们首先将现有目录`pkg / apis / samplecontroller`重命名为`pkg / apis / cnat`，然后创建我们自己的CRD和自定义资源，如下所示：

```

$ cat artifacts / examples / cnat-crd.yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
元数据:
  name: ats.cnat.programming-kubernetes.info
规格:
  group: cnat.programming-kubernetes.info
  版本: v1alpha1
  名称:
    亲切的: 在
    复数: ats
  范围: Namespaced

$ cat artifacts / examples / cnat-example.yaml
apiVersion: cnat.programming-kubernetes.info/v1alpha1
亲切的: 在
元数据:
  标签:
    controller-tools.k8s.io: "1.0"
  name: example-at
规格:
  时间表"2019-04-12T10:12:00Z"
  command::"echo YAY"

```

请注意，每当API类型发生更改时（例如，当您向 `At` CRD 添加新字段时），您必须执行`update-codegen.sh`脚本，如下所示：

```
$ ./hack/update-codegen.sh
```

这将自动生成以下内容：

- 包装的 API / CNAT / v1alpha1 / zz_generated.deepcopy.go
- PKG /生成/*

在业务逻辑方面，我们在运营商中实现了两个部分：

- 在 [types.go](#) 中，我们修改 AtSpec 结构以包含相应的字段，例如 schedule 和 command。请注意， update-codegen.sh 每当您在此处更改某些内容时都必须运行，以便重新生成相关文件。
- 在 [controller.go](#) 中，我们更改 NewController() 和 syncHandler() 函数以及添加辅助函数，包括创建窗格和检查调度时间。

在 [types.go](#) 中，请注意代表 At 资源三个阶段的三个常量：直到预定的时间 PENDING，然后 RUNNING 到完成，最后在 DONE 状态：

```
// +genclient
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

const (
    PhasePending = "PENDING"
    PhaseRunning = "RUNNING"
    PhaseDone     = "DONE"
)

// AtSpec defines the desired state of At
type AtSpec struct {
    // Schedule is the desired time the command is supposed to be executed.
    // Note: the format used here is UTC time https://www.utctime.net
    Schedule string `json:"schedule,omitempty"`
    // Command is the desired command (executed in a Bash shell) to be
    // executed.
    Command string `json:"command,omitempty"`
}

// AtStatus defines the observed state of At
type AtStatus struct {
    // Phase represents the state of the schedule: until the command is
    // executed it is PENDING, afterwards it is DONE.
    Phase string `json:"phase,omitempty"`
}
```

请注意构建标记的显式用法 `+k8s:deepcopy-gen:interfaces`（请参阅[第5章](#)），以便自动生成相应的源。

我们现在可以实现自定义控制器的业务逻辑。也就是说，我们从阶段实现三者之间的状态过渡 PhasePending 到 PhaseRunning 以 PhaseDone -in [controller.go](#)。

在“[工作队列](#)”中，我们介绍并解释了 client-go 提供的工作队列。现在，我们可以把这些知识来工作：在 `processNextWorkItem()` 中 [controller.go](#) -to 更准确地说，在[线路176至186](#) -你可以找到以下（生成）的代码：

```

if when, err := c.syncHandler(key); err != nil {
    c.workqueue.AddRateLimited(key)
    return fmt.Errorf("error syncing '%s': %s, requeuing", key, err.Error())
} else if when != time.Duration(0) {
    c.workqueue.AddAfter(key, when)
} else {
    // Finally, if no error occurs we Forget this item so it does not
    // get queued again until another change happens.
    c.workqueue.Forget(obj)
}

```

这个片段展示了如何 `syncHandler()` 调用我们的（尚未编写的）自定义函数（稍后解释）并涵盖以下三种情况：

1. 第一个 `if` 分支通过 `AddRateLimited()` 函数调用重新排列项目，处理瞬态错误。
2. 第二个分支，`else if` 通过 `AddAfter()` 函数调用重新排列项目以避免热循环。
3. 最后一种情况 `else` 是，该项已成功处理并通过 `Forget()` 函数调用被丢弃。

现在我们已经对通用处理有了充分的了解，让我们继续讨论特定于业务逻辑的功能。关键是上述 `syncHandler()` 功能，我们正在实现自定义控制器的业务逻辑。它有以下签名：

```

// syncHandler compares the actual state with the desired state and attempts
// to converge the two. It then updates the Status block of the At resource
// with the current status of the resource. It returns how long to wait
// until the schedule is due.
func (c *Controller) syncHandler(key string) (time.Duration, error) {
    ...
}

```

此 `syncHandler()` 函数实现以下状态转换：[1](#)

```

...
// If no phase set, default to pending (the initial phase):
if instance.Status.Phase == "" {
    instance.Status.Phase = cnatv1alpha1.PhasePending
}

// Now let's make the main case distinction: implementing
// the state diagram PENDING -> RUNNING -> DONE
switch instance.Status.Phase {
case cnatv1alpha1.PhasePending:
    klog.Infof("instance %s: phase=PENDING", key)
    // As long as we haven't executed the command yet, we need
    // to check if it's time already to act:
    klog.Infof("instance %s: checking schedule %q", key, instance.Spec.Schedule)
    // Check if it's already time to execute the command with a
    // tolerance of 2 seconds:
    d, err := timeUntilSchedule(instance.Spec.Schedule)
}

```

```

if err != nil {
    utilruntime.HandleError(fmt.Errorf("schedule parsing failed: %v", err))
    // Error reading the schedule - requeue the request:
    return time.Duration(0), err
}
klog.Infof("instance %s: schedule parsing done: diff=%v", key, d)
if d > 0 {
    // Not yet time to execute the command, wait until the
    // scheduled time
    return d, nil
}

klog.Infof(
    "instance %s: it's time! Ready to execute: %s", key,
    instance.Spec.Command,
)
instance.Status.Phase = cnatv1alpha1.PhaseRunning
case cnatv1alpha1.PhaseRunning:
    klog.Infof("instance %s: Phase: RUNNING", key)

pod := newPodForCR(instance)

// Set At instance as the owner and controller
owner := metav1.NewControllerRef(
    instance, cnatv1alpha1.SchemeGroupVersion.
    WithKind("At"),
)
pod.ObjectMeta.OwnerReferences = append(pod.ObjectMeta.OwnerReferences, *owner)

// Try to see if the pod already exists and if not
// (which we expect) then create a one-shot pod as per spec:
found, err := c.KubeClientset.CoreV1().Pods(pod.Namespace).
    Get(pod.Name, metav1.GetOptions{})
if err != nil && errors.NotFound(err) {
    found, err = c.KubeClientset.CoreV1().Pods(pod.Namespace).Create(pod)
    if err != nil {
        return time.Duration(0), err
    }
    klog.Infof("instance %s: pod launched: name=%s", key, pod.Name)
} else if err != nil {
    // requeue with error
    return time.Duration(0), err
} else if found.Status.Phase == corev1.PodFailed ||
    found.Status.Phase == corev1.PodSucceeded {
    klog.Infof(
        "instance %s: container terminated: reason=%q message=%q",
        key, found.Status.Reason, found.Status.Message,
    )
    instance.Status.Phase = cnatv1alpha1.PhaseDone
} else {
    // Don't requeue because it will happen automatically
}

```

```

        // when the pod status changes.
        return time.Duration(0), nil
    }
    case cnatv1alpha1.PhaseDone:
        klog.Infof("instance %s: phase: DONE", key)
        return time.Duration(0), nil
    default:
        klog.Infof("instance %s: NOP")
        return time.Duration(0), nil
    }

    // Update the At instance, setting the status to the respective phase:
    _, err = c.cnatClientset.CnatV1alpha1().Ats(instance.Namespace).
        UpdateStatus(instance)
    if err != nil {
        return time.Duration(0), err
    }

    // Don't requeue. We should be reconcile because either the pod or
    // the CR changes.
    return time.Duration(0), nil
}

```

此外，为了设置线人和控制器，我们在以下方面实施以下内容 NewController()：

```

// NewController returns a new cnat controller
func NewController(
    kubeClientset kubernetes.Interface,
    cnatClientset clientset.Interface,
    atInformer informers.AtInformer,
    podInformer corev1informer.PodInformer) *Controller {

    // Create event broadcaster
    // Add cnat-controller types to the default Kubernetes Scheme so Events
    // can be logged for cnat-controller types.
    utilruntime.Must(cnatscheme.AddToScheme(scheme.Scheme))
    klog.V(4).Info("Creating event broadcaster")
    eventBroadcaster := record.NewBroadcaster()
    eventBroadcaster.StartLogging(klog.Infof)
    eventBroadcaster.StartRecordingToSink(&typedcorev1.EventsSinkImpl{
        Interface: kubeClientset.CoreV1().Events("")})
}

source := corev1.EventSource{Component: controllerAgentName}
recorder := eventBroadcaster.NewRecorder(scheme.Scheme, source)

rateLimiter := workqueue.DefaultControllerRateLimiter()
controller := &Controller{
    kubeClientset: kubeClientset,
    cnatClientset: cnatClientset,
    atLister:      atInformer.Lister(),
    atsSynced:     atInformer.Informer().HasSynced,
}

```

```

    podLister:      podInformer.Lister(),
    podsSynced:    podInformer.Informer().HasSynced,
    workqueue:     workqueue.NewNamedRateLimitingQueue(rateLimiter, "Ats"),
    recorder:      recorder,
}

klog.Info("Setting up event handlers")
// Set up an event handler for when At resources change
atInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc: controller.enqueueAt,
    UpdateFunc: func(old, new interface{}) {
        controller.enqueueAt(new)
    },
})
// Set up an event handler for when Pod resources change
podInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
    AddFunc: controller.enqueuePod,
    UpdateFunc: func(old, new interface{}) {
        controller.enqueuePod(new)
    },
})
return controller
}

```

为了使它工作，我们还需要两个辅助函数：一个计算直到计划的时间，如下所示：

```

func timeUntilSchedule(schedule string) (time.Duration, error) {
    now := time.Now().UTC()
    layout := "2006-01-02T15:04:05Z"
    s, err := time.Parse(layout, schedule)
    if err != nil {
        return time.Duration(0), err
    }
    return s.Sub(now), nil
}

```

另一个使用 busybox 容器图像创建一个带有要执行的命令的pod：

```

func newPodForCR(cr *cnatv1alpha1.At) *corev1.Pod {
    labels := map[string]string{
        "app": cr.Name,
    }
    return &corev1.Pod{
        ObjectMeta: metav1.ObjectMeta{
            Name:      cr.Name + "-pod",
            Namespace: cr.Namespace,
            Labels:    labels,
        },
        Spec: corev1.PodSpec{

```

```

Containers: []corev1.Container{
    {
        Name:      "busybox",
        Image:     "busybox",
        Command:   strings.Split(cr.Spec.Command, " "),
    },
},
RestartPolicy: corev1.RestartPolicyOnFailure,
},
}
}

```

我们将 `syncHandler()` 在本章后面的函数中重用这两个辅助函数和业务逻辑的基本流程，因此请确保您熟悉它们的详细信息。

请注意，从 `At` 资源的角度来看，`pod` 是辅助资源，控制器必须确保清理这些 `pod` 或以其他方式冒孤立的 `pod`。

现在，它 `sample-controller` 是学习如何制作香肠的好工具，但通常您希望专注于创建业务逻辑而不是处理样板代码。为此，您可以选择两个相关项目：`Kubebuilder` 和 `Operator SDK`。让我们看看每个以及如何 `cnaat` 实现它们。

Kubebuilder

[Kubebuilder](#)，拥有由 [Kubernetes](#) 特殊兴趣小组（SIG）API Machinery 维护，是一个工具和一套库，使您能够以简单有效的方式构建操作员。Kubebuilder 深度潜水的最佳资源是在线 [Kubebuilder 书籍](#)，它将向您介绍其组件和用法。但是，我们将重点关注 `cnaat` 使用 Kubebuilder 实现我们的运算符（请参阅 [我们的 Git 存储库 中的相应目录](#)）。

首先，让我们确保安装所有依赖项 - 即 `dep`，`kustomize`（参见“[Kustomize](#)”）和 [Kubebuilder 本身](#)：

```

$ dep版本
DEP:
版本: v0.5.1
建造日期: 2019-03-11
去hash : faa6189
go version: go1.12
go编译器: gc
平台: darwin / amd64
特征 : ImportDuringSolve=false

$ kustomize版本
版本: {KustomizeVersion: v2.0.3 GitCommit: a6f65144121d1955266b0cd836ce954c04122dc8
BuildDate: 2019-03-18T22:15:21 +00:00 GoOs: darwin GoArch: amd64}

$ Kubebuilder版本
版本: version.Version {
    KubeBuilder 版本: "1.0.8",
}

```

```

    KubernetesVendor : "1.13.1",
    GitCommit: "1adf50ed107f5042d7472ba5ab50d5e1d357169d";
    BuildDate: "2019-01-25T23:14:29Z", GoOs: "unknown", GoArch: "unknown"
}

```

我们将引导您完成 `cnat` 从头开始编写操作符的步骤。首先，创建一个您选择的目录（我们在我们的仓库中使用`cnat-kubebuilder`），您将用作所有其他命令的基础。

警告

在撰写本文时，Kubebuilder正在转向新版本（v2）。由于它还不稳定，我们会显示（稳定）[版本v1](#)的命令和设置。

引导

至引导 `cnat` 操作符，我们使用这样的 `init` 命令（请注意，这可能需要几分钟，具体取决于您的环境）：

```

$ kubebuilder init \
    --domain programming-kubernetes.info \
    --license apache2 \
    --owner "Programming Kubernetes authors"
运行`dep确保`获取依赖项(推荐的) [y / n ]?
和
保证
跑步.....
使
go generate ./pkg / .... / cmd / ...
go fmt ./pkg / .... / cmd / ...
去兽医./pkg / .... / cmd / ...
去运行vendor / sigs.k8s.io / controller-tools / cmd / controller-gen / main.go all
CRD舱单下产生'config/crds'
RBAC舱单下产生'config/rbac'
去test./pkg / ... ./cmd / ... -coverprofile cover.out
? github.com/mhausenblas/cnat-kubebuilder/pkg/apis [没有test 文件]
? github.com/mhausenblas/cnat-kubebuilder/pkg/controller [没有test 文件]
? github.com/mhausenblas/cnat-kubebuilder/pkg/webhook [没有test 文件]
? github.com/mhausenblas/cnat-kubebuilder/cmd/manager [没有test 文件]
构建-o bin / manager github.com/mhausenblas/cnat-kubebuilder/cmd/manager

```

完成此命令后，Kubebuilder已经为操作员搭建了支架，有效地生成了一堆文件，从自定义控制器到样本CRD。您的现在，基本目录应该类似于以下内容（为清晰起见，不包括巨大的供应商目录）：

```

$ 树 - 我的供应商
.
├── Dockerfile
├── Gopkg.lock
└── Gopkg.toml

```

```

├──Makefile
├──项目
├──宾
| └──经理
| ├──cmd
| └──经理
| └──main.go
├──配置
| ├──cds
| └──默认
|   ├──kustomization.yaml
|   ├──manage_auth_proxy_patch.yaml
|   ├──manage_image_patch.yaml
|   └──manage_prometheus_metrics_patch.yaml
| └──经理
|   └──manage.yaml
└──rbac
  ├──auth_proxy_role.yaml
  ├──auth_proxy_role_binding.yaml
  ├──auth_proxy_service.yaml
  ├──rbac_role.yaml
  └──rbac_role_binding.yaml
└──cover.out
└──黑客
  └──boarplate.go.txt
└──pkg
  ├──apis
  | └──apis.go
  ├──控制器
  | └──controller.go
  └── webhook
    └── webhook.go

```

13目录， 22文件

接下来，我们创建一个API，即一个自定义控制器 - 使用该 `create api` 命令（这应该比上一个命令更快，但仍然需要一点时间）：

```

$ kubebuilder创建api \
    --group cnat \
    --version v1alpha1 \
    - 来吧
在pkg / apis [y / n 下创建资源]?
和
在pkg / controller [y / n 下创建控制器]?
和
写脚手架for你要编辑.....
包装/的API / CNAT / v1alpha1 / at_types.go
包装/的API / CNAT / v1alpha1 / at_types_test.go
PKG /控制器/ AT / at_controller.go

```

```

PKG /控制器/ AT / at_controller_test.go
跑步.....。
go generate ./pkg / .... / cmd / ...
go fmt ./pkg / .... / cmd / ...
去兽医./pkg / .... / cmd / ...
去运行vendor / sigs.k8s.io / controller-tools / cmd / controller-gen / main.go all
CRD舱单下产生'config/crds'
RBAC舱单下产生'config/rbac'
去test./pkg / ... ./cmd / ... -coverprofile cover.out
? github.com/mhausenblas/cnat-kubebuilder/pkg/apis [没有test 文件]
? github.com/mhausenblas/cnat-kubebuilder/pkg/apis/cnat [没有test 文件]
ok github.com/mhausenblas/cnat-kubebuilder/pkg/apis/cnat/v1alpha1 9.011s
? github.com/mhausenblas/cnat-kubebuilder/pkg/controller [没有test 文件]
ok github.com/mhausenblas/cnat-kubebuilder/pkg/controller/at 8.740s
? github.com/mhausenblas/cnat-kubebuilder/pkg/webhook [没有test 文件]
? github.com/mhausenblas/cnat-kubebuilder/cmd/manager [没有test 文件]
构建-o bin / manager github.com/mhausenblas/cnat-kubebuilder/cmd/manager

```

让我们看看发生了什么变化，重点关注已经收到更新和补充的两个目录：

```

$ 树配置/ pkg /
配置/
├─cds
Nat └──cnat_v1alpha1_at.yaml
├─默认
│├─kustomization.yaml
| ├─manage_auth_proxy_patch.yaml
| ├─manage_image_patch.yaml
| └─manage_prometheus_metrics_patch.yaml
├─经理
└─└─manage.yaml
├─rbac
|├─auth_proxy_role.yaml
|├─auth_proxy_role_binding.yaml
|├─auth_proxy_service.yaml
|├─rbac_role.yaml
|└─rbac_role_binding.yaml
└─样品
    └─cnat_v1alpha1_at.yaml
包装/
├─apis
├─addtoscheme_cnat_v1alpha1.go
| ├─apis.go
. └─猫
|├─group.go
|└─v1alpha1
| ├─at_types.go
| ├─at_types_test.go
| ├─doc.go
| ├─register.go

```

```

| |---v1alpha1_suite_test.go
| |---zz_generated.deepcopy.go
| ---控制器
| |---add_at.go
| |---在
| | |---at_controller.go
| | |---at_controller_suite_test.go
| | |---at_controller_test.go
| |---controller.go
| --- webhook
| |---webhook.go

```

11目录， 27文件

请注意在 `config / crds` 中添加了 `cnat_v1alpha1_at.yaml`, 它是CRD, 以及 `config / samples` /中的 `**cnat_v1alpha1_at.yaml` (是, 同名), 表示CRD的自定义资源示例实例。此外, 在 `pkg` /中我们看到了许多新文件, 最重要的是 `apis / cnat / v1alpha1 / at_types.go` 和 `controller / at / at_controller.go`, 我们将在下面修改这两个文件。

接下来, 我们 `cnat` 在Kubernetes中创建一个专用的命名空间并将其用作默认值, 将上下文设置如下 (作为一种好的做法, 总是使用专用的命名空间, 而不是 `default` 一个) :

```
$ kubectl创建ns cnat && \
  kubectl config set-context $(kubectl config current-context )--namespace =cnat
```

我们安装CRD:

```

$ make install
去运行vendor / sigs.k8s.io / controller-tools / cmd / controller-gen / main.go all
根据'config/crds'
RBAC生成的CRD清单生成'config/rbac'
kubectl apply -f config / crds
customresourcedefinition.apiextensions.k8s.io/ats.cnat.programming-kubernetes.info
created

```

和 现在我们可以本地启动运营商:

```

$ 跑步
go generate ./pkg / .... / cmd / ...
go fmt ./pkg / .... / cmd / ...
去兽医./pkg / .... / cmd / ...
去运行./cmd/manager/main.go
{"level": "info", "ts": 1559152740.0550249,  : "logger", "entrypoint":
  "msg": , "setting up client for manager"}
{"level" : 1559152740.057556,  : , :
  : ,  : 1559152740.1396701,  : , :
  : ,  : 1559152740.1397,  : , :
  : ,  : 1559152740.139773,  : , :

```

```

: , : 1559152740.139831, : , :
, :
: : , : 1559152740.139929, : , :
, : , :
"info""ts""logger""entrypoint""msg""setting up manager"}
{"level""info""ts""logger""entrypoint""msg""Registering Components."}
{"level""info""ts""logger""entrypoint""msg""setting up scheme"}
{"level""info""ts""logger""entrypoint""msg""Setting up controller"}
 {"level""info""ts""logger""kubebuilder.controller""msg""Starting EventSource""controller""at-controller""source""kind source: /, Kind=""}
 {"level""info""ts""logger""kubebuilder.controller""msg""Starting EventSource""controller""at-controller""source""kind source: /, Kind=""}
 {"level": "info", "ts": 1559152740.139971, "logger": "entrypoint",
 "msg": "setting up webhooks"}
 {"level": "info", "ts": 1559152740.13998, "logger": "entrypoint",
 "msg": "Starting the Cmd."}
 {"level": "info", "ts": 1559152740.244628, "logger": "kubebuilder.controller",
 "msg": "Starting Controller", "controller": "at-controller"}
 {"level": "info", "ts": 1559152740.344791, "logger": "kubebuilder.controller",
 "msg": "Starting workers", "controller": "at-controller", "worker count": 1}

```

离开终端会话正在运行，并在新会话中安装CRD，验证它，并创建示例自定义资源，如下所示：

```

$ kubectl apply -f config / crds / cnat_v1alpha1_at.yaml
customresourcedefinition.apiextensions.k8s.io/ats.cnat.programming-kubernetes.info
配置

$ kubectl得到crds
姓名创建于
ats.cnat.programming-kubernetes.info 2019-05-29T17: 54: 51Z

$ kubectl apply -f config / samples / cnat_v1alpha1_at.yaml
at.cnat.programming-kubernetes.info/at-sample created

```

如果现在查看 make run 运行的会话的输出，您应该注意以下输出：

```

...
{"level": "info", "ts": 1559153311.659829, : "logger", "controller":
"msg", "Creating Deployment": "namespace", "cnat": "name": , "at-sample-deployment
"}
 {"level": 1559153311.678407, : , :
, : , : , : 1559153311.6839428, : , :
, : , : , : 1559153311.693443, : , :
, : , : , : 1559153311.7023401, : , :
, : , : , : "info""ts""logger""controller""msg""Updating Deployment""namespace""cn
at""name""at-sample-deployment"]
 {"level""info""ts""logger""controller""msg""Updating Deployment""namespace""cnat""n
ame""at-sample-deployment"}
 {"level""info""ts""logger""controller""msg""Updating Deployment""namespace""cnat""n
ame""at-sample-deployment"}
 {"level""info""ts""logger""controller""msg""Updating Deployment""namespace""cnat""n
ame""at-sample-deployment"}

```

```

ame""at-sample-deployment"}
{"level""info""ts""logger""controller""msg""Updating Deployment""namespace""cnat""n
ame""at-sample-deployment"}
{"level""info""ts":1559153332.986961,"logger":"controller",#
 "msg":"Updating Deployment","namespace":"cnat","name":"at-sample-deployment"}

```

这告诉我们整体设置成功！现在我们已经完成了脚手架并成功启动了 `cnat` 运营商，我们可以继续实际的核心任务：`cnat` 使用Kubebuilder实现业务逻辑。

商业逻辑

对于首先，我们将`config / crds / cnat_v1alpha1_at.yaml`和`config / samples / cnat_v1alpha1_at.yaml`更改为我们自己的 `cnat` CRD定义和自定义资源值，重新使用与“Follow sample-controller”相同的结构。

在业务逻辑方面，我们在运营商中实现了两个部分：

- 在`pkg / apis / cnat / v1alpha1 / at_types.go`中，我们修改 `AtSpec` 结构以包含相应的字段，例如 `schedule` 和 `command`。请注意，`make` 每当您在此处更改某些内容时都必须运行，以便重新生成相关文件。Kubebuilder使用Kubernetes生成器（在第5章中描述）并发布它自己的一组生成器（例如，生成CRD清单）。
- 在`pkg / controller / at / at_controller.go`中，我们修改了 `Reconcile(request reconcile.Request)` 在定义的时间创建pod的方法 `Spec.Schedule`。

在`at_types.go`:

```

const (
    PhasePending = "PENDING"
    PhaseRunning = "RUNNING"
    PhaseDone    = "DONE"
)

// AtSpec defines the desired state of At
type AtSpec struct {
    // Schedule is the desired time the command is supposed to be executed.
    // Note: the format used here is UTC time https://www.utctime.net
    Schedule string `json:"schedule,omitempty"`
    // Command is the desired command (executed in a Bash shell) to be executed.
    Command string `json:"command,omitempty"`
}

// AtStatus defines the observed state of At
type AtStatus struct {
    // Phase represents the state of the schedule: until the command is executed
    // it is PENDING, afterwards it is DONE.
    Phase string `json:"phase,omitempty"`
}

```

在 `at_controller.go` 我们落实三个阶段之间的状态转变， PENDING 到 RUNNING 到 DONE :

```

func (r *ReconcileAt) Reconcile(req reconcile.Request) (reconcile.Result, error) {
    reqLogger := log.WithValues("namespace", req.Namespace, "at", req.Name)
    reqLogger.Info("==> Reconciling At")
    // Fetch the At instance
    instance := &cnatv1alpha1.At{}
    err := r.Get(context.TODO(), req.NamespacedName, instance)
    if err != nil {
        if errors.NotFound(err) {
            // Request object not found, could have been deleted after
            // reconcile request--return and don't requeue:
            return reconcile.Result{}, nil
        }
        // Error reading the object--requeue the request:
        return reconcile.Result{}, err
    }

    // If no phase set, default to pending (the initial phase):
    if instance.Status.Phase == "" {
        instance.Status.Phase = cnatv1alpha1.PhasePending
    }

    // Now let's make the main case distinction: implementing
    // the state diagram PENDING -> RUNNING -> DONE
    switch instance.Status.Phase {
    case cnatv1alpha1.PhasePending:
        reqLogger.Info("Phase: PENDING")
        // As long as we haven't executed the command yet, we need to check if
        // it's already time to act:
        reqLogger.Info("Checking schedule", "Target", instance.Spec.Schedule)
        // Check if it's already time to execute the command with a tolerance
        // of 2 seconds:
        d, err := timeUntilSchedule(instance.Spec.Schedule)
        if err != nil {
            reqLogger.Error(err, "Schedule parsing failure")
            // Error reading the schedule. Wait until it is fixed.
            return reconcile.Result{}, err
        }
        reqLogger.Info("Schedule parsing done", "Result", "diff",
            fmt.Sprintf("%v", d))
        if d > 0 {
            // Not yet time to execute the command, wait until the scheduled time
            return reconcile.Result{RequeueAfter: d}, nil
        }
        reqLogger.Info("It's time!", "Ready to execute", instance.Spec.Command)
        instance.Status.Phase = cnatv1alpha1.PhaseRunning
    case cnatv1alpha1.PhaseRunning:
        reqLogger.Info("Phase: RUNNING")
        pod := newPodForCR(instance)
    }
}

```

```

// Set At instance as the owner and controller
err := controllerutil.SetControllerReference(instance, pod, r.scheme)
if err != nil {
    // requeue with error
    return reconcile.Result{}, err
}
found := &corev1.Pod{}
nsName := types.NamespacedName{Name: pod.Name, Namespace: pod.Namespace}
err = r.Get(context.TODO(), nsName, found)
// Try to see if the pod already exists and if not
// (which we expect) then create a one-shot pod as per spec:
if err != nil && errors.NotFound(err) {
    err = r.Create(context.TODO(), pod)
    if err != nil {
        // requeue with error
        return reconcile.Result{}, err
    }
    reqLogger.Info("Pod launched", "name", pod.Name)
} else if err != nil {
    // requeue with error
    return reconcile.Result{}, err
} else if found.Status.Phase == corev1.PodFailed ||
    found.Status.Phase == corev1.PodSucceeded {
    reqLogger.Info("Container terminated", "reason",
        found.Status.Reason, "message", found.Status.Message)
    instance.Status.Phase = cnatv1alpha1.PhaseDone
} else {
    // Don't requeue because it will happen automatically when the
    // pod status changes.
    return reconcile.Result{}, nil
}
case cnatv1alpha1.PhaseDone:
    reqLogger.Info("Phase: DONE")
    return reconcile.Result{}, nil
default:
    reqLogger.Info("NOP")
    return reconcile.Result{}, nil
}

// Update the At instance, setting the status to the respective phase:
err = r.Status().Update(context.TODO(), instance)
if err != nil {
    return reconcile.Result{}, err
}

// Don't requeue. We should be reconcile because either the pod
// or the CR changes.
return reconcile.Result{}, nil
}

```

请注意，最后的 `update` 调用操作在/`status`子资源上（参见“[Status subresource](#)”）而不是整个CR。因此，在这里我们遵循规范状态分割的最佳实践。

现在，一旦 example-at 创建了CR，我们就会看到本地执行的运算符的以下输出：

```

"example-at"
{"level": "info", "ts": "2019-04-12T10:12:00Z", "logger": "controller", "msg": "Phase: PENDING", "namespace": "cnat", "at": "example-at"}
{"level": "info", "ts": "2019-04-12T10:12:00Z", "logger": "controller", "msg": "Checking schedule", "namespace": "cnat", "at": "example-at", "Target": "2019-04-12T10:12:00Z"}
{"level": "info", "ts": 1555063927.492915, "logger": "controller", "msg": "Schedule parsing done", "namespace": "cnat", "at": "example-at", "Result": "2019-04-12 10:12:00 +0000 UTC with a diff of -7.492877s"}
{"level": 1555063927.4929411, "logger": "controller", "msg": "It's time!", "namespace": "cnat", "at": "example-at", "Time": "2019-04-12T10:12:00Z", "Offset": 0}
{"level": 1555063927.626236, "logger": "controller", "msg": "Ready to execute", "namespace": "cnat", "at": "example-at", "Time": "2019-04-12T10:12:00Z", "Offset": 0}
{"level": 1555063927.626303, "logger": "controller", "msg": "==== Reconciling At", "namespace": "cnat", "at": "example-at", "Time": "2019-04-12T10:12:00Z", "Offset": 0}
{"level": "info", "ts": "2019-04-12T10:12:00Z", "logger": "controller", "msg": "Phase: RUNNING", "namespace": "cnat", "at": "example-at"}
{"level": "info", "ts": "2019-04-12T10:12:00Z", "logger": "controller", "msg": "Pod launched", "namespace": "cnat", "at": "example-at", "Name": "example-at-pod", "Time": "2019-04-12T10:12:00Z", "Offset": 0}
{"level": "info", "ts": 1555063928.199562, "logger": "controller", "msg": "==== Reconciling At", "namespace": "cnat", "at": "example-at", "Time": "2019-04-12T10:12:00Z", "Offset": 0}
{"level": "info", "ts": 1555063928.199645, "logger": "controller", "msg": "Phase: DONE", "namespace": "cnat", "at": "example-at", "Time": "2019-04-12T10:12:00Z", "Offset": 0}
{"level": "info", "ts": 1555063937.631733, "logger": "controller", "msg": "==== Reconciling At", "namespace": "cnat", "at": "example-at", "Time": "2019-04-12T10:12:00Z", "Offset": 0}
{"level": "info", "ts": 1555063937.631783, "logger": "controller", "msg": "Phase: DONE", "namespace": "cnat", "at": "example-at", "Time": "2019-04-12T10:12:00Z", "Offset": 0}
...

```

至 验证我们的自定义控制器是否已完成其工作，执行：

```

$ kubectl get pods
NAME READY STATUS RESTARTS AGE
pod / example-at-pod 0/1完成 0 38s

```

大！该 `example-at-pod` 有已创建，现在是时候看到操作的结果：

```

$ kubectl get pod example-at-pod
SUMMER

```

完成自定义控制器的开发后，使用此处所示的本地模式，您可能希望从中构建容器图像。随后可以使用该自定义控制器容器映像，例如，在Kubernetes部署中。您可以使用以下命令生成容器图像并将其推送到repo `quay.io/pk/cnat`:

```
$ export IMG=quay.io/pk/cnat:v1
$ 使docker-build
$ 使docker-push
```

有了这个，我们转向运营商SDK，它共享一些Kubebuilder的代码库和API。

运营商SDK

至为了更容易构建Kubernetes应用程序，CoreOS / Red Hat将运营商框架整合在一起。其中一部分是[Operator SDK](#)，它使开发人员无需深入了解Kubernetes API即可构建运算符。

Operator SDK提供了构建，测试和打包运算符的工具。虽然SDK中有更多可用的功能，特别是在测试时，我们专注于 `cnat` 使用SDK实现我们的运算符（请参阅[我们的Git存储库中的相应目录](#)）。

首先要做的事情是：确保[安装Operator SDK](#)并检查所有依赖项是否可用：

```
$ dep版本
DEP:
版本: v0.5.1
建造日期: 2019-03-11
去hash      : faa6189
go version: go1.12
go编译器: gc
平台: darwin / amd64
特征 : ImportDuringSolve=false

$ operator-sdk --version
operator-sdk version v0.6.0
```

引导

现在是时候 `cnat` 按如下方式引导操作员了：

```
$ operator-sdk new cnat-operator && cd cnat-operator
```

接下来，和Kubebuilder非常相似，我们添加一个API - 或简单地说：初始化自定义控制器，如下所示：

```
$ operator-sdk add api \
--api-version =cnat.programming-kubernetes.info/v1alpha1 \
```

```
--kind =At

$ operator-sdk add controller \
    --api-version =cnat.programming-kubernetes.info/v1alpha1 \
    --kind =At
```

这些命令产生必要的样板代码以及一些辅助功能，如深复印功能 DeepCopy()，DeepCopyInto() 和 DeepCopyObject()。

现在 我们可以将自动生成的CRD应用于Kubernetes集群：

```
$ kubectl apply -f deploy / crds / cnat_v1alpha1_at_crd.yaml  
$ kubectl get crds  
姓名创建于  
ats.cnat.programming-kubernetes.info 2019-04-01T14:03:33Z
```

让我们 cnat 在本地启动我们的自定义控制器。有了它，它可以开始处理请求：

```
$ OPERATOR_NAME=cnatop operator-sdk up local --namespace "cnat"
INFO [0000] 在本地运行运算符。
INFO [0000] 使用命名空间cnat。
{"level": "info", "ts": 1555041531.871706, "logger": "cmd", "msg": "Go Version: go1.12.1"}
{"level": "info", "ts": 1555041531.871785, "logger": "cmd", "msg": "Version of operator-sdk: v0.6.0"}
{"level": "info", "ts": 1555041531.8718028, "logger": "cmd", "msg": "Trying to become the leader."}
{"level": "info", "ts": 1555041531.8739321, "logger": "cmd", "msg": "Skipping leader election; not running in a cluster."}
{"level": "info", "ts": 1555041531.8743382, "logger": "cmd", "msg": "Registering Components."}
{"level": "info", "ts": 1555041536.1611362, "logger": "kubebuilder.controller", "msg": "Starting EventSource", "controller": "at-controller", "source": "kind source: /, Kind="}
{"level": "info", "ts": 1555041536.162519, "logger": "kubebuilder.controller", "msg": "Starting EventSource", "controller": "at-controller", "source": "kind source: /, Kind="}
{"level": "info", "ts": 1555041539.978822, "logger": "metrics", "msg": "Skipping metrics Service creation; not running in a cluster."}
{"level": "info", "ts": 1555041539.978875, "logger": "cmd", "msg": "Starting the Cmd."}
{"level": "info", "ts": 1555041540.179469, "logger": "kubebuilder.controller", "msg": "Starting Controller", "controller": "at-controller"}
{"level": "info", "ts": 1555041540.280784, "logger": "kubebuilder.controller", "msg": "Starting Controller", "controller": "at-controller"}
```

```
"msg": "Starting workers", "controller": "at-controller", "worker count": 1}
```

我们的自定义控制器将保持此状态，直到我们创建CR，`ats.cnat.programming-kubernetes.info`。所以我们这样做：

```
$ cat deploy / crds / cnat_v1alpha1_at_cr.yaml
apiVersion: cnat.programming-kubernetes.info/v1alpha1
亲切的：在
元数据：
  name: example-at
规格：
  时间表"2019-04-11T14:56:30Z"
  command::"echo YAY"

$ kubectl apply -f deploy / crds / cnat_v1alpha1_at_cr.yaml

$ kubectl得到
姓名年龄
at.cnat.programming-kubernetes.info/example-at 54s
```

商业逻辑

在业务逻辑方面，我们在运营商中实现了两个部分：

- 在`pkg/apis/cnat/v1alpha1/at_types.go`中，我们修改 `AtSpec struct`以包含相应的字段，例如 `schedule` 和 `command`，并用于 `operator-sdk generate k8s` 重新生成代码，以及使用 `operator-sdk generate openapi` OpenAPI位的命令。
- 在`pkg/controller/at/at_controller.go`中，我们修改了 `Reconcile(request reconcile.Request)` 在定义的时间创建pod 的方法 `Spec.Schedule`。

更详细地应用于自举代码的更改如下（关注相关位）。在`at_types.go`：

```
// AtSpec defines the desired state of At
// +k8s:openapi-gen=true
type AtSpec struct {
    // Schedule is the desired time the command is supposed to be executed.
    // Note: the format used here is UTC time https://www.utctime.net
    Schedule string `json:"schedule,omitempty"`
    // Command is the desired command (executed in a Bash shell) to be executed.
    Command string `json:"command,omitempty"`
}

// AtStatus defines the observed state of At
// +k8s:openapi-gen=true
type AtStatus struct {
    // Phase represents the state of the schedule: until the command is executed
    // it is PENDING, afterwards it is DONE.
    Phase string `json:"phase,omitempty"`
}
```

```
}
```

在 `at_controller.go` 我们实施了三个阶段的状态图， PENDING 来 RUNNING 来 DONE 。

注意

这 `controller-runtime` 是另一个 SIG API Machinery 拥有的项目，旨在为 Go 包形式的构建控制器提供一套通用的低级功能。有关详细信息，请参阅 [第4章](#)。

由于 Kubebuilder 和 Operator SDK 共享控制器运行时，该 `Reconcile()` 函数实际上是相同的：

```
func (r *ReconcileAt) Reconcile(request reconcile.Request) (reconcile.Result, error
) {
    the-same-as-for-kubebuilder
}
```

`example-at` 创建 CR 后，我们会看到本地执行的运算符的以下输出：

```
$ OPERATOR_NAME=cnatop operator-sdk up local--namespace "cnat"
INFO [0000] 在本地运行运算符。
INFO [0000] 使用命名空间 cnat.

...
{"level": "info", "ts": 1555044934.023597, "logger": "controller_at": "msg", "msg": "==== Reconciling At": "namespace", "cnat": "at": "example-at"} {"level": 1555044934.023713, "logger": "controller_at", "msg": "Phase: PENDING", "namespace": "cnat", "at": "example-at"} {"level": "info", "ts": 1555044934.0237482, "logger": "controller_at", "msg": "Checking schedule", "namespace": "cnat", "at": "example-at", "Target": "2019-04-12T04:56:00Z"} {"level": "info", "ts": 1555044934.02382, "logger": "controller_at", "msg": "Schedule parsing done", "namespace": "cnat", "at": "example-at", "Result": "2019-04-12 04:56:00 +0000 UTC with a diff of 25.976236s"} {"level": "info", "ts": 1555044934.148148, "logger": "controller_at", "msg": "==== Reconciling At", "namespace": "cnat", "at": "example-at"} {"level": "info", "ts": 1555044934.148224, "logger": "controller_at", "msg": "Phase: PENDING", "namespace": "cnat", "at": "example-at"} {"level": 1555044934.148243, "logger": "controller_at", "msg": "Schedule parsing done", "namespace": "cnat", "at": "example-at"} {"level": "info", "ts": 1555044934.1482902, "logger": "controller_at", "msg": "Schedule parsing done", "namespace": "cnat", "at": "example-at", "Target": "2019-04-12T04:56:00Z"} {"level": "info", "ts": 1555044944.1504588, "logger": "controller_at", "msg": "==== Reconciling At", "namespace": "cnat", "at": "example-at", "Target": "2019-04-12T04:56:00Z"} {"level": "info", "ts": 1555044944.150568, "logger": "controller_at", "msg": "Schedule parsing done", "namespace": "cnat", "at": "example-at", "Target": "2019-04-12T04:56:00Z"}
```

```

{"level""info""ts""logger""controller_at""msg""Schedule parsing done""namespace""cnat""at""example-at""Result""2019-04-12 04:56:00 +0000 UTC with a diff of 25.85174s"}
}
{"level""info""ts""logger""controller_at""msg""== Reconciling At""namespace""cnat""at""example-at"}
{"level""info""ts""logger""controller_at""msg""Phase: PENDING""namespace""cnat", "at": "example-at"}
{"level": "info", "ts": 1555044944.150599, : "logger", "controller_at": "msg", "Checking schedule": "namespace", "cnat": "at", "example-at": "Target": , "2019-04-12T04:56:00Z"}
{"level" : 1555044944.150663, : , :
, : , : ,
: , : 1555044954.385175, : ,
, : , : , : 1555044954.3852649, : ,
, : , : , : 1555044954.385288, : ,
, "info""ts""logger""controller_at""msg""Schedule parsing done""namespace""cnat""at""example-at""Result""2019-04-12 04:56:00 +0000 UTC with a diff of 15.84938s"}
{"level""info""ts""logger""controller_at""msg""== Reconciling At""namespace""cnat""at""example-at"}
{"level""info""ts""logger""controller_at""msg""Phase: PENDING""namespace""cnat""at""example-at"}
{"level""info""ts""logger""controller_at""msg""Checking schedule""namespace": "cnat", "at": "example-at",
"Target": "2019-04-12T04:56:00Z"}
{"level": "info", "ts": 1555044954.38534, : "logger", "controller_at": "msg", "Schedule parsing done": "namespace", "cnat": "at", "example-at": "Result": , "2019-04-12 04:56:00 +0000 UTC with a diff of 5.614691s"}
{"level" : 1555044964.518383, : , :
, : , : , : 1555044964.5184839, : ,
, : , : , : 1555044964.518566, : ,
: , : , : ,
: : , : 1555044964.5186381, : "info""ts""logger""controller_at""msg""== Reconciling At""namespace""cnat""at""example-at"]
{"level""info""ts""logger""controller_at""msg""Phase: PENDING""namespace""cnat""at""example-at"}
{"level""info""ts""logger""controller_at""msg""Checking schedule""namespace""cnat""at""example-at""Target""2019-04-12T04:56:00Z"}
{"level""info""ts""logger""controller_at",
"msg": "Schedule parsing done", "namespace": "cnat", "at": "example-at",
"Result": "2019-04-12 04:56:00 +0000 UTC with a diff of -4.518596s"}
{"level": "info", "ts": 1555044964.5186849, : "logger", "controller_at": "msg", "It's time!": "namespace", "cnat": "at", "example-at": "Ready to execute": , "echo YAY"}
{"level" : 1555044964.642559, : , :
, : , : , : 1555044964.642622, : ,
, : , : , : 1555044964.911037 , : ,
: , : , : , : 1555044964.9111192, "info""ts""logger""controller_at""msg""== Reconciling At""namespace""cnat""at""example-at"]
{"level""info""ts""logger""controller_at""msg""Phase: RUNNING""namespace""cnat""at""example-at"}
{"level""info""ts""logger""controller_at""msg""== Reconciling At""namespace""cnat"

```

```

"at""example-at"]
{"level""info""ts""logger":"controller_at",
 "msg":"Phase: RUNNING", "namespace":"cnat", "at":"example-at"}
 {"level":"info", "ts":1555044966.038684, "logger":"controller_at",
  "msg":">== Reconciling At", "namespace":"cnat", "at":"example-at"}
 {"level":"info", "ts":1555044966.038771, "logger":"controller_at",
  "msg":"Phase: DONE", "namespace":"cnat", "at":"example-at"}
 {"level":"info", "ts":1555044966.708663, "logger":"controller_at",
  "msg":">== Reconciling At", "namespace":"cnat", "at":"example-at"}
 {"level":"info", "ts":1555044966.708749, "logger":"controller_at",
  "msg":"Phase: DONE", "namespace":"cnat", "at":"example-at"}
 ...

```

在这里你可以看到我们的运营商的三个阶段： PENDING 直到时间戳 1555044964.518566 ，然后 RUNNING ，然后 DONE 。

要验证自定义控制器的功能并检查操作结果，请输入：

```

$ kubectl到达, 豆英
姓名年龄
at.cnat.programming-kubernetes.info/example-at 23m

NAME READY STATUS RESTARTS AGE
pod / example-at-pod 0/1已完成 0 46秒

$ kubectl记录example-at-pod
SUMMER

```

完成自定义控制器的开发后，使用此处所示的本地模式，您可能希望从中构建容器图像。随后可以使用该自定义控制器容器映像，例如，在Kubernetes部署中。您可以使用以下命令生成容器图像：

```
$ operator-sdk build $REGISTRY/ PROJECT / IMAGE
```

这里 是了解更多有关Operator SDK及其示例的更多资源：

- Toader Sebastian在BanzaiCloud上发表的“[Kubernetes Operator SDK完整指南](#)”
- Rob Szumski的博客文章“[为Prometheus和Thanos建立一个Kubernetes运营商](#)”
- 来自Cloudark on ITNEXT的“[改善可用性的Kubernetes运营商开发指南](#)”

为了总结本章，我们来看一些编写自定义控制器和运算符的替代方法。

其他方法

在 除了我们讨论过的方法之外，或者可能与之相结合，您可能需要查看以下项目，库和工具：

- [Metacontroller](#)

该Metacontroller的基本思想是为您提供状态和变化的声明性规范，与JSON接口，基于级别触发的协调循环。也就是说，您将收到描述观察状态的JSON并返回描述所需状态的JSON。这对于在Python或JavaScript等动态脚本语言中快速开发自动化特别有用。除了简单的控制器，Metacontroller还允许您将API组合成更高级别的抽象 - 例如，[BlueGreenDeployment](#)。

- [EVERWHERE](#)

类似对于Metacontroller，KUDO提供了一种声明性方法来构建Kubernetes运算符，涵盖整个应用程序生命周期。简而言之，它是Mesosphere从Apache Mesos框架体验到Kubernetes的经验。KUDO高度自以为是，但也易于使用，几乎不需要编码；实质上，您必须指定的是Kubernetes清单的集合，其中包含用于定义何时执行的内置逻辑。

- [Rook操作工具包](#)

这个是一个实现运营商的通用库。它起源于Rook运营商，但已被分拆成一个独立的独立项目。

- [ericchiang / K8S](#)

这个是使用Kubernetes协议缓冲支持生成的Eric Chiang精简的Go客户端。它的行为类似于官方的Kubernetes `client-go`，但只导入两个外部依赖项。虽然它有一些限制 - 例如，在[集群访问配置方面](#) - 它是一个简单易用的Go包。

- [kutil](#)

AppsCode通过提供Kubernetes `client-go` 附加组件 `kutil`。

- 基于CLI客户端的方法

一个客户端方法，主要用于实验和测试，是以 `kubectl` 编程方式利用（例如，[kubecuddler](#)库）。

注意

虽然我们专注于使用本书中的Go编程语言编写运算符，但您可以使用其他语言编写运算符。二值得注意的例子是Flant的[Shell-operator](#)，它使您能够在良好的旧shell脚本中编写运算符，以及Zalando的[Kopf \(Kubernetes运算符框架\)](#)，Python框架和库。

正如本章开头所提到的，运营商领域正在迅速发展，越来越多的从业者以代码和最佳实践的形式分享他们的知识，因此请关注这里的新工具。请一定要看看网上资源和论坛，比如 `#kubernetes-operators`，`#kubebuilder` 和 `#client-go-docs` 渠道上Kubernetes松弛，学习新的方法和/或讨论问题，当你被卡住得到帮助。

吸收和未来方向

评委们仍然认为编写运算符的方法将是最受欢迎和广泛使用的。在Kubernetes项目的背景下，在CR和控制器方面，有几个SIG的活动。主要利益相关者是SIG [API Machinery](#)，它拥有CR和控制器，负责[Kubebuilder](#)项目。运营商SDK已经加大了与Kubebuilder API的协调力度，因此存在很多重叠。

摘要

在本章中，我们了解了不同的工具，使您可以更有效地编写自定义控制器和运算符。传统上，跟随它 `sample-controller` 是唯一的选择，但是使用Kubebuilder和Operator SDK，您现在有两个选项可以让您专注于自定义控制器的业务逻辑而不是处理样板。幸运的是，这两个工具共享了很多API和代码，因此从一个工具转移到另一个工具应该不会太困难。

现在，让我们看看如何提供我们的劳动成果 - 即如何打包和运送我们一直在编写的控制器。

¹我们只在这里展示相关部分; 函数本身有很多其他的样板代码，我们并不关心它们。

第7章运输控制器和操作员

现在您已经熟悉了自定义控制器的开发，接下来我们将讨论如何使您的自定义控制器和操作员准备就绪的主题。在本章中，我们将讨论控制器和操作员的操作方面，向您展示如何打包它们，引导您完成在生产中运行控制器的最佳实践，并确保您的扩展点不会破坏您的Kubernetes集群，安全性，或表现方面。

生命周期管理和包装

在本节我们考虑运营商的生命周期管理。也就是说，我们将讨论如何打包和运送您的控制器或操作员，以及如何处理升级。当您准备将运营商发送给用户时，您需要一种方法来安装它。为此，您需要打包相应的工件，例如定义控制器二进制文件的YAML清单（通常作为Kubernetes部署），以及CRD和安全相关资源，例如服务帐户和必要的RBAC权限。一旦您的目标用户运行某个版本的运营商，您还需要有一个机制来升级控制器，考虑版本控制和潜在的零停机升级。

让我们从最简单的结果开始：打包和交付控制器，以便用户可以直接安装它。

包装：挑战

而Kubernetes定义带有清单的资源，通常用YAML编写，一个声明资源状态的低级接口，这些清单文件都有缺点。最重要的是在包装容器化应用程序的背景下，YAML清单是静态的；也就是说，YAML清单中的所有值都是固定的。这意味着，如果要更改[部署清单](#)中的容器映像，则必须创建新清单。

让我们看一个具体的例子。假设您在名为`mycontroller.yaml`的YAML清单中编码了以下Kubernetes部署，表示您希望用户安装的自定义控制器：

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mycustomcontroller
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: customcontroller
    spec:
      containers:
        - name: thecontroller
          image: example/controller:0.1.0
          ports:
            - containerPort: 9999
          env:
            - name: REGION
              value: eu-west-1
```

想象一下，环境变量 `REGION` 定义了控制器的某些运行时属性，例如托管服务网格等其他服务的可用性。换句话说，虽然默认值 `eu-west-1` 可能是合理的，但用户可以并且可能会根据自己的偏好或策略覆盖它。

现在，鉴于YAML清单`mycontroller.yaml`本身是一个静态文件，其中包含在编写时定义的所有值 - 并且客户端（例如 `kubectl` 本身不支持清单中的变量部分） - 如何使用户能够提供变量值或在运行时覆盖现有值？也就是说，在前面的例子中，用户可以设置 `REGION` 为，例如，`us-east-2` 何时使用它们进行安装（例如）`kubectl apply`？

克服构建时间，静态的这些限制 YAML 在 Kubernetes 中显示，有一些选项可以模拟清单（例如 Helm）或以其他方式启用变量输入（Kustomize），具体取决于用户提供的值或运行时属性。

舵

[头盔](#)，哪个自称是 Kubernetes 的软件包经理，最初由 Deis 开发，现在是一个云计算本地计算基金会（CNCF）项目，主要贡献者来自微软、谷歌和 Bitnami（现在是 VMware 的一部分）。

舵通过定义和应用所谓的图表，有效地参数化 YAML 清单，帮助您安装和升级 Kubernetes 应用程序。以下是[示例图表模板](#)的摘录：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "flagger.fullname" . }}
...
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app.kubernetes.io/name: {{ template "flagger.name" . }}
      app.kubernetes.io/instance: {{ .Release.Name }}
  template:
    metadata:
      labels:
        app.kubernetes.io/name: {{ template "flagger.name" . }}
        app.kubernetes.io/instance: {{ .Release.Name }}
  spec:
    serviceAccountName: {{ template "flagger.serviceAccountName" . }}
    containers:
      - name: flagger
        securityContext:
          readOnlyRootFilesystem: true
          runAsUser: 10001
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

如您所见，变量以 `{{ .Some.value.here }}` 格式编码，恰好是[Go 模板](#)。

要安装图表，可以运行该 `helm install` 命令。虽然Helm有多种查找和安装图表的方法，但最简单的方法是使用官方稳定图表之一：

```
# get the latest list of charts:
$ 掌舵回购更新

# install MySQL:
$ helm install stable / mysql
释放微笑企鹅

# list running apps:
$ 掌舵ls
名称版本更新状态图表
微笑 - 企鹅 1           星期三9月2812:59:46 2016 部署mysql-0.1.0

# remove it:
$ 掌舵删除微笑 - 企鹅
去掉了微笑的企鹅
```

为了打包控制器，您需要为它创建一个Helm图表并将其发布到某个地方，默认情况下将其发布到索引并可通过[Helm Hub](#)访问的公共存储库，[如图7-1所示](#)。

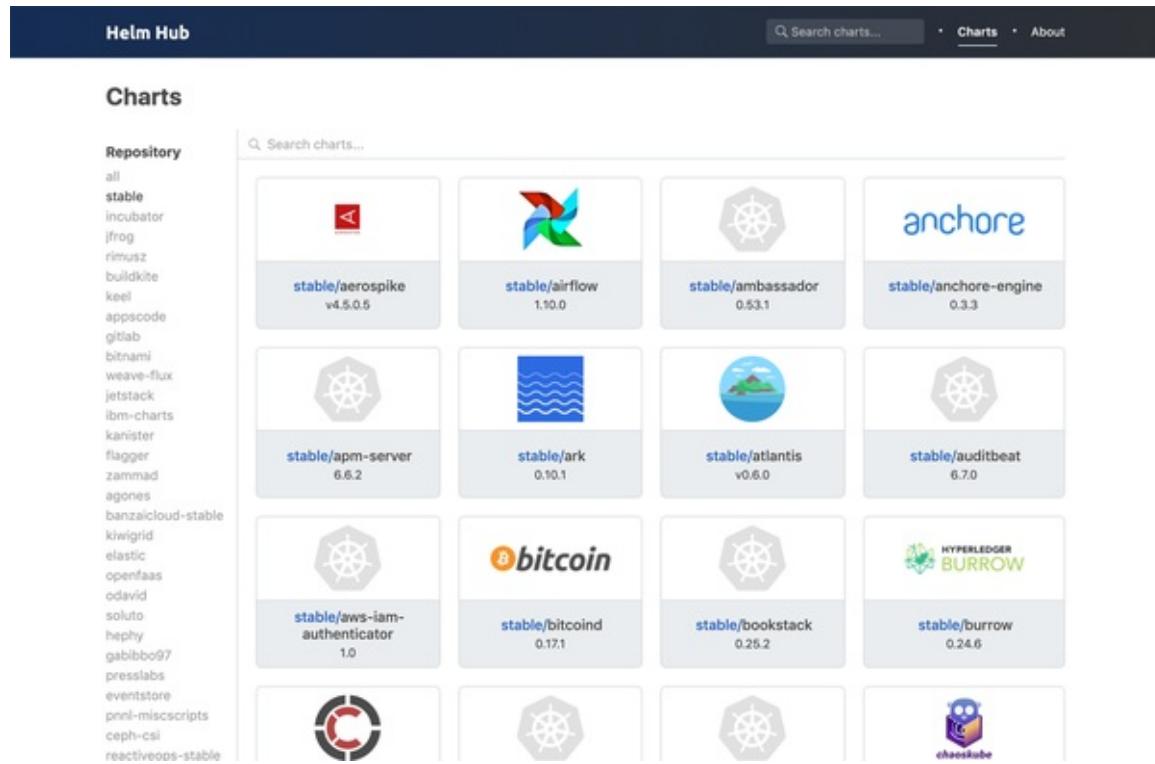


图7-1。Helm Hub屏幕截图显示了公开的Helm图表

有关如何创建Helm图表的进一步指导，请在闲暇时仔细阅读以下资源：

- Bitnami的优秀文章[“如何创建你的第一个头盔图”](#)。
- “如果要将图表保留在自己的组织中，请[使用S3作为Helm存储库](#)”。

- Helm官方文档：“[图表最佳实践指南](#)”。

Helm很受欢迎，部分原因是它最终用户易于使用。然而，一些人认为目前的Helm架构带来了[安全风险](#)。好消息是社区正在积极致力于解决这些问题。

Kustomize

[Kustomize](#)提供Kubernetes清单文件配置自定义的声明性方法，遵循熟悉的Kubernetes API。它于[2018年中期推出](#)，现在是Kubernetes SIG CLI项目。

你可以[安装](#)你的机器上Kustomize，作为一个独立的，或者，如果你有一个较新的 `kubectl` 版本（比1.14更新），它被[运与](#) `kubectl` 和与激活的 `-k` 命令行标志。

因此，Kustomize允许您自定义原始YAML清单文件，而无需触及原始清单。但是这在实践中如何运作？我们假设您要打包我们的 `cnat` 自定义控制器；你定义了一个名为`kustomize.yaml`的文件，它看起来像：

```
imageTags:
  - name: quay.io/programming-kubernetes/cnat-operator
    newTag: 0.1.0
resources:
  - cnat-controller.yaml
```

现在，您可以将此应用于`cnav-controller.yaml`文件，例如，使用以下内容：

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: cnat-controller
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: cnat
    spec:
      containers:
        - name: custom-controller
          image: quay.io/programming-kubernetes/cnat-operator
```

使用 `kustomize build` 和保留`cnav-controller.yaml`文件不变！ - 然后输出：

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: cnat-controller
spec:
  replicas: 1
```

```

template:
  metadata:
    labels:
      app: cnat
  spec:
    containers:
      - name: custom-controller
        image: quay.io/programming-kubernetes/cnat-operator:0.1.0

```

该 `kustomize build` 例如，`can` 的输出可以在 `kubectl apply` 命令中使用，并自动为您应用所有[自定义](#)。

有关 Kustomize 的更详细的演练以及如何使用它，请查看以下资源：

- Sébastien Goasguen 的博文“[使用 kustomize 配置 Kubernetes 应用程序](#)”。
- Kevin Davin 的帖子“[Kustomize 在 Kubernetes 做模板的正确方法](#)”。
- 视频“[TGI Kubernetes 072: Kustomize 和朋友](#)”，在那里你可以看到 Joe Beda 应用它。

鉴于 Kustomize 的原生支持，`kubectl` 越来越多的用户可能会采用它。请注意，虽然它解决了一些问题（自定义），但生命周期管理的其他方面（如验证和升级）可能需要您将 Kustomize 与 Google 的 [CUE](#) 等语言一起使用。

为了总结这个包装主题，让我们回顾一下从业者使用的其他一些解决方案。

其他包装选项

一些上述包装选择的显着替代品 - 以及[野外](#) 的许多其他选择：

- **UNIX 工具**

在为了定制原 Kubernetes 舱单的值，你可以使用一系列的 CLI 工具，如 `sed`，`awk` 或 `jq` 在 shell 脚本。这是一种流行的解决方案，至少在 Helm 到来之前，可能是最广泛使用的选项 - 尤其是因为它最大限度地减少了依赖性，并且在 *nix 环境中相当便携。

- **传统的配置管理系统**

You 可以使用任何传统的配置管理系统（例如 Ansible、Puppet、Chef 或 Salt）来打包和交付您的操作员。

- **云本地语言**

一个新一代所谓的[云原生编程语言](#)，如 Pulumi 和 Ballerina，允许 Kubernetes 原生应用程序的打包和生命周期管理等。

- [平均总评分](#)

和 `ytt` 你在一起还有一个 YAML 模板工具的另一个选项，它使用的语言本身就是 Google 配置语言 [Starlark](#) 的修改版本。它在 YAML 结构上进行语义操作，并侧重于可重用性。

- [Ksonnet](#)

一个用于Kubernetes清单的配置管理工具，最初由Heptio（现在的VMware）开发，Ksonnet已被弃用，并且不再积极工作，因此使用它需要您自担风险。

阅读更多关于Jesse Suen的帖子“[Kubernetes配置管理状态：一个未解决的问题](#)”中讨论的选项。

现在我们已经讨论了一般的包装选项，让我们来看看包装和运输控制器和操作员的最佳实践。

包装最佳实践

什么时候打包并发布您的运营商，确保您了解以下最佳做法。无论您选择哪种机制（Helm，Kustomize，shell脚本等），这些都适用：

- 提供适当的访问控制设置：这意味着在最小权限的基础上为控制器定义专用服务帐户以及RBAC权限；有关详细信息，请参阅[“获得权限”](#)。
- 考虑自定义控制器的范围：它会在一个命名空间或多个命名空间中查看CR吗？查看[Alex Ellis关于不同方法的利弊的Twitter对话](#)。
- 测试并对控制器进行分析，以便了解其占用空间和可扩展性。例如，Red Hat已经将一组详细的要求与OperatorHub [贡献](#)指南中的说明放在一起。
- 使确保CRD和控制器都有详细记录，最好使用[godoc.org](#)上提供的内联文档和一组用法示例；请参阅Banzai Cloud的[银行金库](#)运营商获取灵感。

生命周期管理

一个与包/船相比，更广泛和更全面的方法是生命周期管理。基本思路是考虑整个供应链，从开发到运输再到升级，并尽可能自动化。在这个领域，CoreOS（以及后来的Red Hat）再次成为开拓者：应用相同的逻辑，导致运营商进行生命周期管理。换句话说：为了安装并稍后升级操作员的自定义控制器，您将拥有一个专门的操作员，知道如何处理操作员。实际上，运营商框架的一部分 - 也提供了运营商SDK，如“[运营商SDK](#)”中所讨论的 - 是所谓的[运营商生命周期管理器（OLM）](#)。

Jimmy Zelinskie是OLM背后的主要人物之一，其措辞如下：

OLM为操作员作者做了很多工作，但它也解决了一个很少有人想过的重要问题：你如何有效地管理Kubernetes的一流扩展？

简而言之，OLM提供了一种声明性的方式来安装和升级运营商及其依赖项，以及Helm等互补包装解决方案。如果您想购买成熟的OLM解决方案或为版本控制和升级挑战创建临时解决方案，这取决于您；但是，你应该在这里制定一些策略。对于某些领域 - 例如，Red Hat对运营商中心的[认证过程](#) - 它不仅是推荐的，而且对于任何重要的部署方案都是强制性的，即使您没有针对Hub。

生产就绪部署

在本节我们将回顾并讨论如何使您的自定义控制器和操作员准备好生产。以下是高级别清单：

- 使用Kubernetes [部署](#)或DaemonSet来监督您的自定义控制器，以便它们在失败时自动重启 - 并且会失败。
- 实行通过专用端点进行健康检查以获得活跃度和准备度探测。这与上一步一起使您的操作更具弹

性。

- 考虑领导者 - 追随者/待命模型，以确保即使您的控制器pod崩溃，其他人也可以接管。但请注意，同步状态是一项非常重要的任务。
- 提供访问控制资源，例如服务帐户和角色，应用最小特权原则；有关详细信息，请参阅[“获取权限”](#)。
- 考虑自动构建，包括测试。“[自动构建和测试](#)”中提供了更多提示。
- 主动处理监测和记录；有关内容和方式，请参阅[“自定义控制器和可观察性”](#)。

我们还建议您仔细阅读上述文章[“提高可用性的Kubernetes运营商开发指南”](#)以了解更多信息。

获得权限

您的自定义控制器是Kubernetes控制平面的一部分。它需要读取资源状态，在Kubernetes内部以及（可能）之外创建资源，并传达其自身资源的状态。对于所有这些，自定义控制器需要一组正确的权限，通过一组基于角色的访问控制（RBAC）相关设置表示。正确的做法是本节的主题。

首先要做的事情：始终创建一个[专用服务帐户](#)来运行您的控制器。换句话说：永远不要 `default` 在命名空间中使用服务帐户。¹

为了使您的生活更轻松，您可以 `clusterRole` 使用必要的RBAC规则定义a 以及 `RoleBinding` 将其绑定到特定命名空间，从而有效地重用跨命名空间的角色，如[使用RBAC授权](#)条目中所述。

以下最小特权原则，仅分配控制器执行其工作所需的权限。例如，如果控制器仅管理pod，则无需为其提供列出或创建部署或服务的权限。此外，请确保控制器不安装CRD和/或准入webhook。换句话说，控制器不应具有管理CRD和 webhook的权限。

如[第6章所述](#)，用于创建自定义控制器的通用工具通常提供用于生成开箱即用的RBAC规则的功能。例如，Kubebuilder生成[以下](#) RBAC资产以及运算符：

```
$ ls与rbac /
共40
drwx ----- 7mhausenblas工作人员 224 124月09:52。
drwx ----- 7mhausenblas工作人员 224 124月09:55 ..
-rw ----- 1mhausenblas工作人员 280 124月09:49 auth_proxy_role.yaml
-rw ----- 1mhausenblas工作人员 257 124月09:49 auth_proxy_role_binding.yaml
-rw ----- 1mhausenblas工作人员 449 124月09:49 auth_proxy_service.yaml
-rw-r - r-- 1mhausenblas工作人员 1044 124月10:50 rbac_role.yaml
-rw-r - r-- 1mhausenblas工作人员 287 124月10:50 rbac_role_binding.yaml
```

查看自动生成的RBAC角色和绑定显示了细粒度的设置。在`rbac_role.yaml`中，您可以找到：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: null
  name: manager-role
rules:
  - apiGroups:
```

```

- apps
resources:
- deployments
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
- apiGroups:
- apps
resources:
- deployments/status
verbs: ["get", "update", "patch"]
- apiGroups:
- cnat.programming-kubernetes.info
resources:
- ats
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
- apiGroups:
- cnat.programming-kubernetes.info
resources:
- ats/status
verbs: ["get", "update", "patch"]
- apiGroups:
- admissionregistration.k8s.io
resources:
- mutatingwebhookconfigurations
- validatingwebhookconfigurations
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
- apiGroups:
- ""
resources:
- secrets
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
- apiGroups:
- ""
resources:
- services
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

```

看看Kubebuilder生成的这些权限 `v1`，你可能会有点吃惊。[2](#)我们当然是：最佳实践告诉我们，如果一个控制器没有很好的理由，它应该不能：

- 通常，写入仅在代码中读取的资源。例如，如果你只是看服务和部署，不要删除 `create`，`update`，`patch`，和 `delete` 中的作用动词。
- 获取所有秘密；也就是说，始终将此限制为必要的最小秘密集。
- 写 `MutatingWebhookConfigurations` 或 `ValidatingWebhookConfigurations`。这相当于访问群集中的任何资源。
- 写 `CustomResourceDefinition`s。请注意，在刚刚显示的集群角色中不允许这样做，但重要的是在此提及：CRD创建应该由单独的进程完成，而不是由控制器本身完成。
- 编写未管理的外部资源的/`status`子资源（请参阅[“子资源”](#)）。例如，此处的部署不由 `cnat` 控制器管理，不应在范围内。

当然，Kubebuilder实际上无法理解您的控制器代码实际上在做什么。因此，生成的RBAC规则过于宽松也就不足为奇了。我们建议按照前面的检查表仔细检查权限并将其减少到绝对最小值。

警告

有读取权限对系统中的所有机密信息赋予控制器访问所有服务帐户令牌的权限。这相当于可以访问群集中的所有密码。具有写访问权

限 `MutatingWebhookConfigurations` 或 `ValidatingWebhookConfigurations` 允许您拦截和操作系统中的每个API请求。这为Kubernetes集群中的rootkit打开了大门。两者都显然非常危险，并被认为是反模式，因此最好避免使用它们。

为了避免拥有太多的权力 - 即将访问权限限制为绝对必要的权限 - 请考虑使用[audit2rbac](#)。此工具使用审核日志生成一组适当的权限，从而实现更安全的设置和更少的麻烦。

从`rbac_role_binding.yaml`您可以了解到：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  creationTimestamp: null
  name: manager-rolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: manager-role
subjects:
- kind: ServiceAccount
  name: default
  namespace: system
```

有关RBAC及其工具的更多最佳实践，请访问[RBAC.dev](#)，这是一个致力于Kubernetes RBAC的网站。现在让我们继续讨论自定义控制器的测试和性能考虑因素。

自动构建和测试

如在云原生土地上的最佳实践，考虑自定义控制器的自动构建。这通常称为连续构建或持续集成（CI），包括单元测试，集成测试，构建容器映像，甚至可能进行健全性或烟雾测试。云原生计算基金会（CNCF）维护着许多可用的开源CI工具的交互式列表。

在构建控制器时，请记住它应尽可能少地消耗计算资源，同时尽可能多地为客户端提供服务。每个CR，基于您定义的CRD，是客户端的代理。但是，你怎么知道它消耗了多少，它是否泄漏记忆，以及它的扩展程度如何？

一旦自定义控制器的开发稳定，您就可以并且确实应该执行大量测试。这些可包括以下内容，但可能不限于此：

- 与Kubernetes本身以及kboom工具相关的性能相关测试可以为您提供有关扩展和资源占用空间的数据。

- 浸泡测试，例如，使用那些在Kubernetes -aim在长期的使用情况，从几个小时到几天，随着揭幕资源的任何泄漏，如文件或主存储器的目标。

作为最佳实践，这些测试应该是CI管道的一部分。换句话说，从第一天开始自动构建自定义控制器，测试和打包。对于一个具体的示例设置，我们鼓励您查看MarkoMudrinić的优秀帖子“[在CI中为集成和E2E测试产生Kubernetes集群](#)”。

接下来，我们将介绍为有效排除故障提供基础的最佳实践：内置支持可观察性。

自定义控制器和可观察性

在本节中，我们将介绍自定义控制器的可观察性方面，特别是日志记录和监视。

记录

使确保您提供足够的日志信息以帮助进行[故障排除](#)（在生产中）。像往常一样在容器化设置中，记录日志信息发送到 `stdout`，可以使用 `kubectl logs` 命令或以聚合形式在每个pod上使用它。可以使用特定于云提供商的解决方案提供聚合，例如Google Cloud中的Stackdriver或AWS中的CloudWatch，或者Elasticsearch-Logstash-Kibana / Elasticsearch-Fluentd-Kibana堆栈等定制解决方案。另见SébastienGoasguen和Michael Hausenblas（O'Reilly）关于此主题的食谱的[Kubernetes Cookbook](#)。

让我们看一下 `cnat` 自定义控制器日志的示例摘录：

```
{
  "level": "info",
  "ts": 1555063927.492718,
  "logger": "controller",
  "msg": "==== Reconciling At" }
{
  "level": "info",
  "ts": 1555063927.49283,
  "logger": "controller",
  "msg": "Phase: PENDING" }
{
  "level": "info",
  "ts": 1555063927.492857,
  "logger": "controller",
  "msg": "Checking schedule" }
{
  "level": "info",
  "ts": 1555063927.492915,
  "logger": "controller",
  "msg": "Schedule parsing done" }
```

该如何记录的：在一般情况下，我们更喜欢[结构化记录](#)和可调日志级别，至少 `debug` 和 `info`。在 Kubernetes代码库中广泛使用了两种方法，除非你有充分的理由不这样做，否则你应该考虑使用它们：

- 的 `logger` 接口，例如，如在发现[httplog.go](#)，与一个具体类型（沿 `respLogger`）-捕获之类的状态和错误。

- `klog`，一把叉子谷歌的 `glog`，是整个Kubernetes使用的结构化记录器，虽然它有它的特性，但值得了解。

在什么样的日志记录：一定要为你的业务逻辑操作的正常情况下，详细的日志信息。例如，从 `cnat` 控制器的Operator SDK实现，在[at_controller.go](#)中，设置记录器，如下所示：

```
reqLogger := log.WithValues("namespace", request.Namespace, "at", request.Name)
```

然后在业务逻辑中，在 `Reconcile(request reconcile.Request)` 函数中：

```
case cnatv1alpha1.PhasePending:
    reqLogger.Info("Phase: PENDING")
    // As long as we haven't executed the command yet, we need to check if it's
    // already time to act:
    reqLogger.Info("Checking schedule", "Target", instance.Spec.Schedule)
    // Check if it's already time to execute the command with a tolerance of
    // 2 seconds:
    d, err := timeUntilSchedule(instance.Spec.Schedule)
    if err != nil {
        reqLogger.Error(err, "Schedule parsing failure")
        // Error reading the schedule. Wait until it is fixed.
        return reconcile.Result{}, err
    }
    reqLogger.Info("Schedule parsing done", "Result", "diff", fmt.Sprintf("%v", d))
    if d > 0 {
        // Not yet time to execute the command, wait until the scheduled time
        return reconcile.Result{RequeueAfter: d}, nil
    }
    reqLogger.Info("It's time!", "Ready to execute", instance.Spec.Command)
    instance.Status.Phase = cnatv1alpha1.PhaseRunning
```

这个Go片段让您了解要记录的内容，尤其是何时使用 `reqLogger.Info` 和 `reqLogger.Error`。

通过使用Logging 101，让我们继续讨论相关主题：指标！

监控，仪表和审计

一个[Prometheus](#)是一个很好的开源，容器就绪的监控解决方案，可以跨环境（内部部署和云端）使用。对每个事件发出警报是不切实际的，因此您可能想要考虑谁需要了解哪种事件。例如，您可以制定一个策略，即由基础架构管理员处理与节点相关的事件或与命名空间相关的事件，并为命名空间级别的事件分页命名空间管理员或开发人员。在这种情况下，为了可视化您收集的指标，最受欢迎的解决方案当然是[Grafana](#)；有关在Grafana中可视化的Prometheus指标的示例，请参见图7-2，该示例取自[Prometheus](#)文档。

如果您正在使用服务网格 - 例如，基于[Envoy代理](#)（如Istio或App Mesh）或Linkerd，那么检测通常 是免费的，或者可以通过最少的（配置）工作来实现。否则，您必须使用相应的库（例如[Prometheus](#)提供的库）来自行公开代码中的相关指标。在这种情况下，您可能还想查看2019年初推出的初出茅庐的

服务网状接口（SMI）项目，该项目旨在为基于CR和控制器的服务网格提供标准化接口。



图7-2。在Grafana中可视化的Prometheus指标

另一个Kubernetes通过API服务器提供的有用功能是[审计](#)，它允许您记录影响群集的一系列活动。审计策略中提供了不同的策略，从无记录到记录事件元数据，请求正文和响应正文。您可以选择简单的日志后端和使用webhook与第三方系统集成。

摘要

本章重点介绍如何通过讨论控制器和操作员的操作方面（包括包装，安全性和性能）使您的操作员准备好生产。

有了这个，我们已经介绍了编写和使用自定义Kubernetes控制器和运算符的基础知识，所以现在我们转向另一种扩展Kubernetes的方法：开发自定义API服务器。

[1](#)另请参阅Luc Juggery的帖子“[Kubernetes Tips: Using a ServiceAccount](#)”，详细讨论服务帐户的使用情况。

[2](#)但是，我们确实针对Kubebuilder项目提出了[问题748](#)。

第8章自定义API服务器

如作为CustomResourceDefinitions的替代方案，您可以使用自定义API服务器。自定义API服务器可以使用与主Kubernetes API服务器相同的方式为API组提供资源。与CRD相比，使用自定义API服务器几乎没有任何限制。

本章首先列出了CRD可能不适合您的用例的一些原因。它描述了聚合模式，可以使用自定义API服务器扩展Kubernetes API表面。最后，您将学习使用Golang实际实现自定义API服务器。

自定义API服务器的用例

一个可以使用自定义API服务器代替CRD。它可以完成CRD可以做的所有事情，并提供几乎无限的灵活性。当然，这需要付出代价：开发和运营的复杂性。

让我们看一下截至本文撰写时CRD的一些限制（当Kubernetes 1.14是稳定版本时）。会议厅文件：

- 用 etcd 他们的存储介质（或任何Kubernetes API服务器使用）。
- 不支持protobuf，只支持JSON。
- 仅支持两种子资源：/status和/scale（参见“[子资源](#)”）。
- 不支持优雅删除。[1](#)终结器可以模拟此但不允许自定义优雅的删除时间。
- 显着增加Kubernetes API服务器的CPU负载，因为所有算法都以通用方式实现（例如，验证）。
- 仅为API端点实现标准CRUD语义。
- 做不支持同居资源（即不同API组中的资源或共享存储的不同名称的资源）。[2](#)

一个相反，自定义API服务器没有这些限制。自定义API服务器：

- 可以使用任何存储介质。有自定义API服务器，例如：
 - 所述[度量API服务器](#)，其存储数据在内存中的最大性能
 - API服务器镜像[OpenShift](#)中的Docker注册表
 - API服务器写入时间序列数据库
 - API服务器镜像云API
 - API服务器镜像其他API对象，例如[OpenShift](#)中反映Kubernetes命名空间的项目
- 可以像所有本地Kubernetes资源一样提供protobuf支持。为此，您必须使用[go-to-protobuf](#)创建一个.proto文件，然后使用protobuf编译器生成序列化程序，然后将其编译为二进制文件。`protoc`
- 可以提供任何自定义子资源；例如，Kubernetes API服务器提供/`exec`, `/logs`, `/port-forward`等，其中大多数使用非常自定义的协议，如WebSockets或HTTP / 2流。
- 可以像Kubernetes对pod一样实现优雅删除。`kubectl`等待删除，用户甚至可以提供定制优雅的终止期。
- 可以使用Golang以最有效的方式实现所有操作，如验证，准入和转换，而无需通过webhook往返，这会增加进一步的延迟。这对于高性能用例或者存在大量对象很重要。想想拥有数千个节点的巨大集群中的pod对象，以及两个数量级的pod。
- 可以实现自定义语义，例如核心v1 `Service` 类型中的服务IP的原子预留。在创建服务时，分配并直接返回唯一的服务IP。在某种程度上，像这样的特殊语义当然可以通过准入webhooks来实现（参见“[Admission Webhooks](#)”），尽管这些webhooks永远无法可靠地知道传递的对象是否实际创

建或更新：它们被乐观地调用，但是稍后请求管道中的步骤可能会取消请求。换句话说： webhook中的副作用很棘手，因为如果请求失败，则没有撤消触发器。

- 可以提供具有公共存储机制（即，公共 etcd 密钥路径前缀）的资源，但是存在于不同的API组中或者以不同的方式命名。例如，Kubernetes在API组中存储部署和其他资源 `extensions/v1`，然后将它们移动到更具体的API组 `apps/v1`。

换句话说，自定义API服务器是CRD仍然有限的情况下的解决方案。在转换到新语义时不要破坏资源兼容性的过渡场景中，自定义API服务器通常更加灵活。

示例：比萨餐厅

至了解自定义API服务器的实现方式，在本节中，我们将介绍一个示例项目：实现比萨餐厅API的自定义API服务器。我们来看看要求。

我们想在 `restaurant.programming-kubernetes.info` API组中创建两种类型：

- `Topping`
披萨配料（例如萨拉米香肠，马苏里拉奶酪或番茄）
- `Pizza`
餐厅提供的比萨饼类型

浇头是集群范围的资源，仅保留一个单位顶部成本的浮点值。一个实例很简单：

```
apiVersion: restaurant.programming-kubernetes.info/v1alpha1
kind: Topping
metadata:
  name: mozzarella
spec:
  cost: 1.0
```

每个披萨都可以有任意数量的配料；例如：

```
apiVersion: restaurant.programming-kubernetes.info/v1alpha1
kind: Pizza
metadata:
  name: margherita
spec:
  toppings:
    - mozzarella
    - tomato
```

排序列表是有序的（与YAML或JSON中的任何列表一样），但顺序对于类型的语义并不重要。无论如何，顾客都会得到相同的披萨。我们希望在列表中允许重复，以便允许比如带有额外奶酪的比萨饼。

所有这些都可以通过CRD轻松实现。现在让我们添加一些超出基本CRD功能的要求：[3](#)

- 我们希望仅允许具有相应 `Topping` 对象的披萨规格中的浇头。
- 我们还想假设我们首先将此API作为 `v1alpha1` 版本引入，但最终了解到我们希望 `v1beta1` 在同一API 的版本中另外表示浇头。

换句话说，我们希望有两个版本并在它们之间无缝转换。

可以在[本书的GitHub存储库中](#)找到此API作为自定义API服务器的完整实现。在本章的其余部分，我们将介绍该项目的所有主要部分并了解其工作原理。在这个过程中，您将在不同的视角中看到前一章中介绍的许多概念：即Klangnetes API服务器后面的Golang实现。CRD中强调的一些设计决策也将变得更加清晰。

因此，即使您不打算使用自定义API服务器的路径，我们也强烈建议您仔细阅读本章。也许这里提出的概念将来也可用于CRD，在这种情况下，了解自定义API服务器将对您有用。

架构：聚合

之前进入技术实现细节，我们希望在Kubernetes集群的上下文中获取自定义API服务器体系结构的更高级别视图。

自定义API服务器是为API组提供服务的进程，通常使用通用API服务器库k8s.io/apiserver构建。这些进程可以在集群内部或外部运行。在前一种情况下，它们在pods内部运行，前面有一项服务。

该称为主Kubernetes API服务器 `kube-apiserver` 始终是 `kubectl` 其他API客户端的第一联系人。由自定义API服务器提供服务的API组由 `kube-apiserver` 进程代理到自定义API服务器进程。换句话说，该 `kube-apiserver` 进程知道所有自定义API服务器及其服务的API组，以便能够向它们代理正确的请求。

该执行此代理的组件位于 `kube-apiserver` 进程内部并被调用 `kube-aggregator`。将API请求代理到自定义API服务器的过程称为*API聚合*。

让我们看一下针对自定义API服务器的请求路径，但是进入Kubernetes API服务器TCP套接字（参见[图8-1](#)）：

1. Kubernetes API服务器收到请求。
2. 它们通过处理程序链，包括身份验证，审计日志记录，模拟，最大限度飞行限制，授权等（图中只是一个草图，并不完整）。
3. 由于Kubernetes API服务器知道聚合API，因此它可以拦截对HTTP路径/`apis /aggregated-API-group-name`请求。
4. Kubernetes API服务器将请求转发给自定义API服务器。

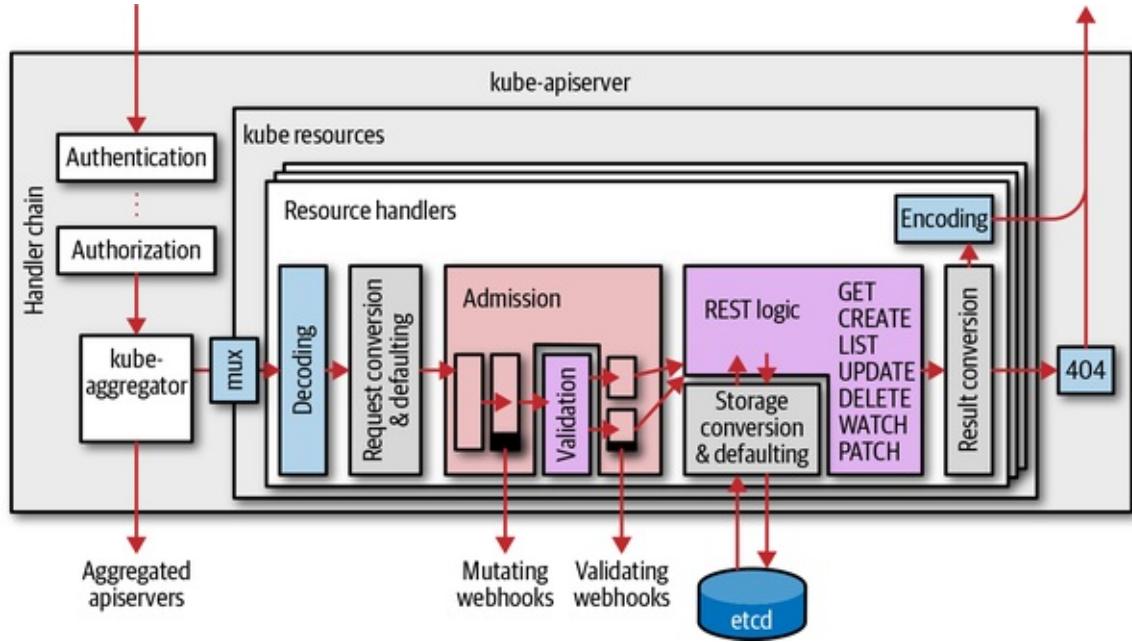


图8-1。Kubernetes主要的API服务器kube-apiserver，带有集成的kube-aggregator

所述 kube-aggregator 的HTTP路径下代理请求的API组版本（即，一切下的`/apis /group-name/version`）。它不必知道API组版本中实际提供的资源。

相反，它 kube-aggregator 为所有聚合的自定义API服务器本身提供发现端点`/apis`和`/apis /group-name`（它使用下一节中解释的已定义顺序）并返回结果，而不与聚合的自定义API服务器通信。相反，它使用来自 APIService 资源的信息。让我们详细看一下这个过程。

API服务

对于要知道自定义API服务器所服务的API组的Kubernetes API服务器，APIService 必须在 `apiregistration.k8s.io/v1` API组中创建一个对象。这些对象仅列出API组和版本，而不是资源或任何进一步的详细信息：

```

apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: name
spec:
  group: API-group-name
  version: API-group-version
  service:
    namespace: custom-API-server-service-namespace
    name: -API-server-service
  caBundle: base64-caBundle
  insecureSkipTLSVerify: bool
  groupPriorityMinimum: 2000
  versionPriority: 20

```

该名称是任意的，但为了清楚起见，我们建议您使用标识API组名称和版本的名称 - 例如，*group-name-version*。

该服务可以是群集中的普通 [ClusterIP 服务](#)，也可以是具有 `ExternalName` 群集外自定义API服务器的给定DNS名称的服务。在这两种情况下，端口必须是443.不支持其他服务端口（在撰写本文时）。服务目标端口映射允许任何选定的，优选非限制的更高端口用于自定义API服务器pod，因此这不是主要限制。

证书颁发机构 (CA) 捆绑包用于Kubernetes API服务器以信任所联系的服务。请注意，API请求可以包含机密数据。为避免中间人攻击，强烈建议您设置 `caBundle` 字段而不使用 `insecureSkipTLSVerify` 替代方法。这对于任何生产群集尤其重要，包括证书轮换机制。

最后，`APIService` 对象中有两个优先级。这些有一些棘手的语义，在Golang代码文档中描述了 `APIService` 类型：

```
// GroupPriorityMinimum is the priority this group should have at least. Higher
// priority means that the group is preferred by clients over lower priority ones.
// Note that other versions of this group might specify even higher
// GroupPriorityMinimum values such that the whole group gets a higher priority.
//
// The primary sort is based on GroupPriorityMinimum, ordered highest number to
// lowest (20 before 10). The secondary sort is based on the alphabetical
// comparison of the name of the object (v1.bar before v1.foo). We'd recommend
// something like: *.k8s.io (except extensions) at 18000 and PaaSes
// (OpenShift, Deis) are recommended to be in the 2000s
GroupPriorityMinimum int32 `json:"groupPriorityMinimum"`

// VersionPriority controls the ordering of this API version inside of its
// group. Must be greater than zero. The primary sort is based on
// VersionPriority, ordered highest to lowest (20 before 10). Since it's inside
// of a group, the number can be small, probably in the 10s. In case of equal
// version priorities, the version string will be used to compute the order
// inside a group. If the version string is "kube-like", it will sort above non
// "kube-like" version strings, which are ordered lexicographically. "Kube-like"
// versions start with a "v", then are followed by a number (the major version),
// then optionally the string "alpha" or "beta" and another number (the minor
// version). These are sorted first by GA > beta > alpha (where GA is a version
// with no suffix such as beta or alpha), and then by comparing major version,
// then minor version. An example sorted list of versions:
// v10, v2, v1, v11beta2, v10beta3, v3beta1, v12alpha1, v11alpha2, foo1, foo10.
VersionPriority int32 `json:"versionPriority"
```

换句话说，该 `GroupPriorityMinimum` 值确定组优先级的位置。如果 `APIService` 不同版本的多个对象不同，则最高值规则。

第二个优先级只是在彼此之间对版本进行排序，以定义动态客户端要使用的首选版本。

以下是 `GroupPriorityMinimum` 本机Kubernetes API组的值列表：

```

var apiVersionPriorities = map[schema.GroupVersion]priority{
    {Group: "", Version: "v1"}: {group: 18000, version: 1},
    {Group: "extensions", Version: "v1beta1"}: {group: 17900, version: 1},
    {Group: "apps", Version: "v1beta1"}: {group: 17800, version: 1},
    {Group: "apps", Version: "v1beta2"}: {group: 17800, version: 9},
    {Group: "apps", Version: "v1"}: {group: 17800, version: 15},
    {Group: "events.k8s.io", Version: "v1beta1"}: {group: 17750, version: 5},
    {Group: "authentication.k8s.io", Version: "v1"}: {group: 17700, version: 15},
    {Group: "authentication.k8s.io", Version: "v1beta1"}: {group: 17700, version: 9},
    {Group: "authorization.k8s.io", Version: "v1"}: {group: 17600, version: 15},
    {Group: "authorization.k8s.io", Version: "v1beta1"}: {group: 17600, version: 9},
    {Group: "autoscaling", Version: "v1"}: {group: 17500, version: 15},
    {Group: "autoscaling", Version: "v2beta1"}: {group: 17500, version: 9},
    {Group: "autoscaling", Version: "v2beta2"}: {group: 17500, version: 1},
    {Group: "batch", Version: "v1"}: {group: 17400, version: 15},
    {Group: "batch", Version: "v1beta1"}: {group: 17400, version: 9},
    {Group: "batch", Version: "v2alpha1"}: {group: 17400, version: 9},
    {Group: "certificates.k8s.io", Version: "v1beta1"}: {group: 17300, version: 9},
    {Group: "networking.k8s.io", Version: "v1"}: {group: 17200, version: 15},
    {Group: "networking.k8s.io", Version: "v1beta1"}: {group: 17200, version: 9},
    {Group: "policy", Version: "v1beta1"}: {group: 17100, version: 9},
    {Group: "rbac.authorization.k8s.io", Version: "v1"}: {group: 17000, version: 15},
    {Group: "rbac.authorization.k8s.io", Version: "v1beta1"}: {group: 17000, version: 12},
    {Group: "rbac.authorization.k8s.io", Version: "v1alpha1"}: {group: 17000, version: 9},
    {Group: "settings.k8s.io", Version: "v1alpha1"}: {group: 16900, version: 9},
    {Group: "storage.k8s.io", Version: "v1"}: {group: 16800, version: 15},
    {Group: "storage.k8s.io", Version: "v1beta1"}: {group: 16800, version: 9}
}

```

```

    sion: 9},
      {Group: "storage.k8s.io", Version: "v1alpha1"}:           {group: 16800, ver
    sion: 1},
      {Group: "apiextensions.k8s.io", Version: "v1beta1"}:       {group: 16700, ver
    sion: 9},
      {Group: "admissionregistration.k8s.io", Version: "v1"}:     {group: 16700, ver
    sion: 15},
      {Group: "admissionregistration.k8s.io", Version: "v1beta1"}: {group: 16700, ver
    sion: 12},
      {Group: "scheduling.k8s.io", Version: "v1"}:             {group: 16600, ver
    sion: 15},
      {Group: "scheduling.k8s.io", Version: "v1beta1"}:          {group: 16600, ver
    sion: 12},
      {Group: "scheduling.k8s.io", Version: "v1alpha1"}:          {group: 16600, ver
    sion: 9},
      {Group: "coordination.k8s.io", Version: "v1"}:            {group: 16500, ver
    sion: 15},
      {Group: "coordination.k8s.io", Version: "v1beta1"}:         {group: 16500, ver
    sion: 9},
      {Group: "auditregistration.k8s.io", Version: "v1alpha1"}:   {group: 16400, ver
    sion: 1},
      {Group: "node.k8s.io", Version: "v1alpha1"}:                {group: 16300, ver
    sion: 1},
      {Group: "node.k8s.io", Version: "v1beta1"}:                 {group: 16300, ver
    sion: 9},
    }
}

```

因此，使用 2000 类似PaaS的API意味着将它们放在此列表的末尾。⁴

API组的顺序在REST映射过程中起作用 `kubectl`（请参阅[“REST映射”](#)）。这意味着它对用户体验有实际影响。如果存在冲突的资源名称或短名称，则具有最高 `GroupPriorityMinimum` 值的名称将获胜。

此外，在使用自定义API服务器替换API组版本的特殊情况下，可能会使用此优先级排序。例如，您可以通过将自定义API服务放在 `GroupPriorityMinimum` 值低于上表中的位置的位置，将原生 Kubernetes API组替换为已修改的组（无论出于何种原因）。

再次注意，Kubernetes API服务器不需要知道发现端点/`apis`和/`apis /group-name`任何资源列表，也不需要知道代理。资源列表仅通过第三个发现端点/`apis /group-name/`返回`version`。但正如我们在上一节中看到的那样，此端点由聚合的自定义API服务器提供，而不是由 `kube-aggregator`。

自定义API服务器的内部结构

一个自定义API服务器类似于组成Kubernetes API服务器的大多数部分，当然具有不同的API组实现，并且没有嵌入式 `kube-aggregator` 或嵌入式 `apiextension-apiserver`（为CRD提供服务）。这导致几乎相同的架构图（如图8-2所示）与[图8-1中](#)的相似：

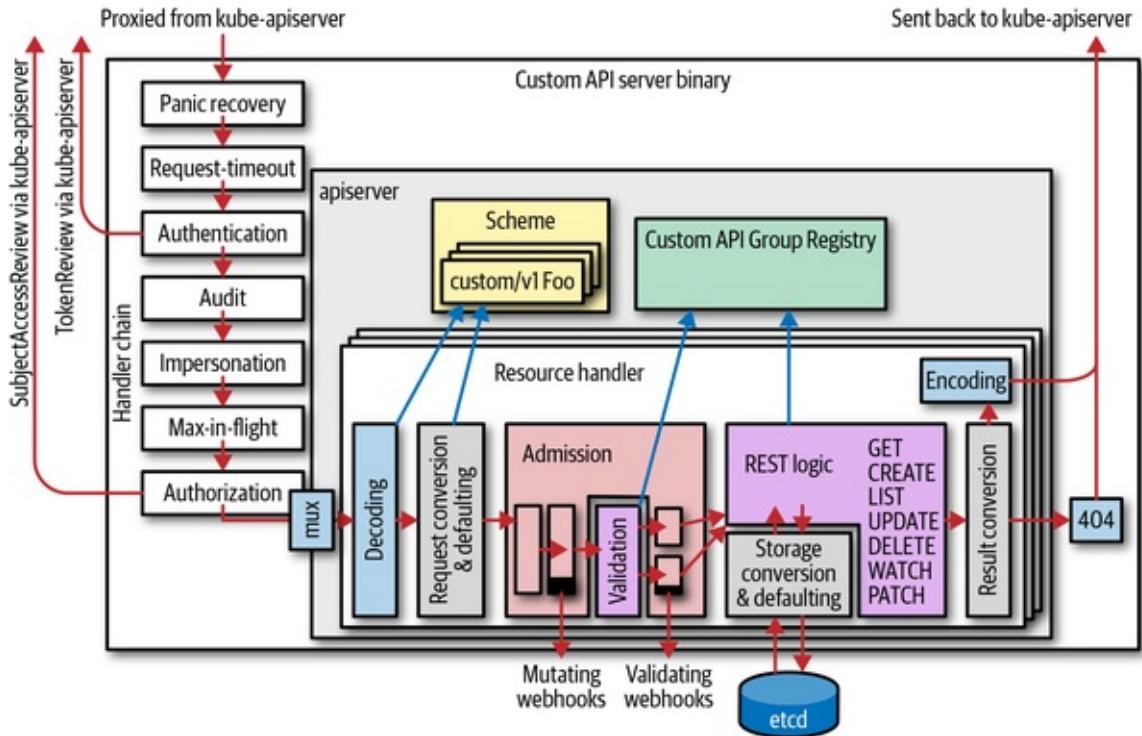


图8-2。基于k8s.io/apiserver的聚合自定义API服务器

我们观察了很多事情。聚合API服务器：

- 具有与Kubernetes API服务器相同的基本内部结构。
- 有它自己的处理程序链，包括身份验证，审计，模拟，最大限度飞行限制和授权（我们将在本章中解释为什么这是必要的；例如，参见“[委托授权](#)”）。
- 有自己的资源处理程序管道，包括解码，转换，许可，REST映射和编码。
- 呼叫入场webhooks。
- 可能会写入 etcd（但它可以使用不同的存储后端）。etcd 群集不必与Kubernetes API服务器使用的群集相同。
- 有自己的自定义API组的方案和注册表实现。注册表实现可能有所不同，并可在任何程度上进行定制
- 再次验证。它通常执行客户端证书身份验证和基于令牌的身份验证，并通过 TokenAccessReview 请求回调Kubernetes API服务器。我们将在稍后更详细地讨论身份验证和信任体系结构。
- 有自己的审计。这意味着Kubernetes API服务器会审核某些字段，但仅限于元级别。对象级审计在聚合的自定义API服务器中完成。
- 使用 SubjectAccessReview 对Kubernetes API服务器的请求进行自己的身份验证。我们将在稍后详细讨论授权。

委托身份验证和信任

一个聚合自定义API服务器（基于[k8s.io/apiserver](#)）构建在与Kubernetes API服务器相同的身份验证库上。它可以使用客户端证书或令牌来验证用户身份。

由于聚合的自定义API服务器在架构上位于Kubernetes API服务器后面（即，Kubernetes API服务器接收请求并将它们代理到聚合的自定义API服务器），因此Kubernetes API服务器已对请求进行了身份验证。所述Kubernetes API服务器存储的认证，也就是说，用户名和组成员的HTTP请求报头，通常的结果 `X-Remote-User` 和 `X-Remote-Group`（这些可以用配置 `--requestheader-username-headers` 和 `--requestheader-group-headers` 标志）。

聚合的自定义API服务器必须知道何时信任这些标头；否则，任何其他呼叫者可以声称已完成身份验证并可以设置这些标头。这由特殊请求标头客户端CA处理。它存储在配置映射 `kube-system / extension-apiserver-authentication` (filename `requestheader-client-ca-file`) 中。这是一个例子：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: extension-apiserver-authentication
  namespace: kube-system
data:
  client-ca-file: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
  requestheader-allowed-names: '["aggregator"]'
  requestheader-client-ca-file: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
  requestheader-extra-headers-prefix: '["X-Remote-Extra-"]'
  requestheader-group-headers: '["X-Remote-Group"]'
  requestheader-username-headers: '["X-Remote-User"]'
```

使用此信息，具有默认设置的聚合自定义API服务器将进行身份验证：

- 使用与给定 `client-ca` 文件匹配的客户端证书的客户端
- 由 Kubernetes API 服务器预先验证的客户端，其请求使用给定的 `requestheader-client-ca-file` 转发，其用户名和组成员身份存储在给定的 HTTP 头中 `X-Remote-Group`，`X-Remote-User`

最后但并非最不重要的是，有一种称为 `TokenAccessReview` 前锋的机制承载令牌（通过 HTTP 头接收）返回 Kubernetes API 服务器以验证它们是否有效。默认情况下禁用令牌访问查看机制，但可以选择启用它；请参阅“[选项和配置模式以及启动管道](#)”。`Authorization: bearer *token*`

我们将在以下部分中看到如何实际设置委派身份验证。虽然我们在这里详细介绍了这种机制，但在聚合的自定义API服务器中，这主要由 `k8s.io/apiserver` 库自动完成。但是知道幕后发生的事情肯定是有价值的，特别是在涉及安全的情况下。

委托授权

后身份验证已完成，每个请求都必须经过授权。授权基于用户名和组列表。该 Kubernetes 中的默认授权机制是基于角色的访问控制（RBAC）。

RBAC将身份映射到角色，将角色映射到授权规则，最终接受或拒绝请求。我们不会详细介绍RBAC授权对象（如角色和集群角色，角色绑定和集群角色绑定）的所有详细信息（有关详细信息，请参阅[“获取权限”](#)）。从架构的角度来看，知道聚合的自定义API服务器通过 `SubjectAccessReview`s 授权使用委托授权就足够了。它不会评估RBAC规则本身，而是将评估委托给Kubernetes API服务器。

为什么聚合API服务器总是需要执行另一个授权步骤

Kubernetes API服务器收到并转发到聚合自定义API服务器的每个请求都会通过身份验证和授权（请参见[图8-1](#)）。这意味着聚合的自定义API服务器可以跳过此类请求的委派授权部分。

但这种预授权不能保证，可能会消失在任何时候（有计划地分割 `kube-aggregator`，从 `kube-apiserver` 更好的安全性，并在未来更好的扩展性）。此外，直接发送到聚合自定义API服务器的请求（例如，通过客户端证书或令牌访问查看进行身份验证）不会通过Kubernetes API服务器，因此未经过预授权。

换句话说，跳过委托授权会打开一个安全漏洞，因此非常不鼓励。

现在让我们更详细地看一下委托授权。

一个主题访问审核是根据请求从聚合的自定义API服务器发送到Kubernetes API服务器（如果它在其授权缓存中找不到答案）。以下是此类评论对象的示例：

```
apiVersion: authorization.k8s.io/v1
kind: SubjectAccessReview
spec:
  resourceAttributes:
    group: apps
    resource: deployments
    verb: create
    namespace: default
    version: v1
    name: example
  user: michael
  groups:
    - system:authenticated
    - admins
    - authors
```

Kubernetes API服务器从聚合的自定义API服务器接收此信息，评估集群中的RBAC规则，并做出决定，返回一个 `SubjectAccessReview` 设置了状态字段的对象；例如：

```
apiVersion: authorization.k8s.io/v1
kind: SubjectAccessReview
status:
  allowed: true
  denied: false
  reason: "rule foo allowed this request"
```

这里注意，有可能是两个 `allowed` 和 `denied` 是 `false`。这意味着Kubernetes API服务器无法做出决定，在这种情况下，聚合自定义API服务器中的另一个授权者可以做出决定（API服务器实现逐个查询的授权链，委派授权是授权者之一在那个链中）。这可用于建模非标准授权逻辑 - 即，在某些情况下，如果没有RBAC规则，而是使用外部授权系统。

请注意，出于性能原因，委派授权机制在每个聚合自定义API服务器中维护本地缓存。默认情况下，它缓存1,024个授权条目：

- 5 允许的授权请求的分钟到期时间
- 30 被拒绝的授权请求的秒到期

这些值可以通过 `--authorization-webhook-cache-authorized-ttl` 和自定义 `--authorization-webhook-cache-unauthorized-ttl`。

我们将在以下部分中看到如何在代码中设置委派授权。同样，与身份验证一样，在聚合的自定义API服务器内部委托授权主要由 `k8s.io/apiserver` 库自动完成。

编写自定义API服务器

在前面的部分中，我们研究了聚合API服务器的体系结构。在本节中，我们将了解Golang中聚合的自定义API服务器的实现。

主要的Kubernetes API服务器是通过 `k8s.io/apiserver` 库实现的。自定义API服务器将使用完全相同的代码。主要区别在于我们的自定义API服务器将在集群中运行。这意味着它可以假设 `kube-apiserver` 群集中有 `a` 可用，并使用它来执行委派授权并检索其他 `kube-native` 资源。

我们还假设 `etcd` 群集可用并准备好由聚合的自定义API服务器使用。这 `etcd` 是专用的还是与 Kubernetes API服务器共享并不重要。我们的自定义API服务器将使用不同的 `etcd` 密钥空间来避免冲突。

本章中的代码示例引用了[GitHub上的示例代码](#)，因此请查看完整的源代码。我们将在此处仅显示最有趣的摘录，但您可以随时查看完整的示例项目，对其进行实验，并且非常重要的学习 - 在真实的群集中运行它。

该 `pizza-apiserver` 项目实现了[“示例：比萨餐厅”](#) 中显示的示例API。

选项和配置模式和启动管道

1. 该 `k8s.io/apiserver` 库使用选项和配置模式来创建正在运行的API服务器。

我们将从几个绑定到标志的选项结构开始。从 `k8s.io/apiserver` 获取它们并添加我们的自定义选项。来自 `k8s.io/apiserver` 的选项结构可以在特殊用例的代码中进行调整，并且提供的标志可以应用于标志集以便用户可访问。

在这个[例子中](#)，我们非常简单地将所有内容都基于 `RecommendedOptions`。这些推荐选项为简单API的“普通”聚合自定义API服务器所需的一切设置，如下所示：

```
import (
```

```

    ...
    informers "github.com/programming-kubernetes/pizza-apiserver/pkg/
    generated/informers/externalversions"
)

const defaultEtcdPathPrefix = "/registry/restaurant.programming-kubernetes.info"

type CustomServerOptions struct {
    RecommendedOptions *genericoptions.RecommendedOptions
    SharedInformerFactory informers.SharedInformerFactory
}

func NewCustomServerOptions(out, errOut io.Writer) *CustomServerOptions {
    o := &CustomServerOptions{
        RecommendedOptions: genericoptions.NewRecommendedOptions(
            defaultEtcdPathPrefix,
            apiserver.Codecs.LegacyCodec(v1alpha1.SchemeGroupVersion),
            genericoptions.NewProcessInfo("pizza-apiserver", "pizza-apiserver"),
        ),
    }
    return o
}

```

该 `CustomServerOptions` 嵌入 `RecommendedOptions` 和顶部添加一个字段。`NewCustomServerOptions` 是 `CustomServerOptions` 使用默认值填充结构的构造函数。

让我们看看一些更有趣的细节：

- `defaultEtcdPathPrefix` 是 etcd 我们所有键的前缀。作为关键空间，我们使用`/registry/pizza-apiserver.programming-kubernetes.info`，明显不同于Kubernetes键。
- `SharedInformerFactory` 是我们自己的CR的全流程共享informer工厂，以避免相同资源的不必要的informer（参见图3-5）。请注意，它是从我们项目中生成的informer代码导入的，而不是从 `client-go`。
- `NewRecommendedOptions` 为具有默认值的聚合自定义API服务器设置所有内容。

我们来看看 `NewRecommendedOptions`：

```

return &RecommendedOptions{
    Etcd:           NewEtcdOptions(storagebackend.NewDefaultConfig(prefix, codec)),
    SecureServing: sso.WithLoopback(),
    Authentication: NewDelegatingAuthenticationOptions(),
    Authorization: NewDelegatingAuthorizationOptions(),
    Audit:          NewAuditOptions(),
    Features:       NewFeatureOptions(),
    CoreAPI:        NewCoreAPIOptions(),
    ExtraAdmissionInitializers:
        func(c *server.RecommendedConfig) ([]admission.PluginInitializer, error) {
            return nil, nil
}

```

```

    },
    Admission:      NewAdmissionOptions(),
    ProcessInfo:    processInfo,
    Webhook:        NewWebhookOptions(),
}

```

如有必要，所有这些都可以调整。例如，如果需要自定义默认服务端口，则 `RecommendedOptions.SecureServing.SecureServingOptions.BindPort` 可以进行设置。

让我们简要介绍一下现有的内容选项结构：

- `Etcd` 配置读写的存储堆栈 `etcd`。
- `SecureServing` 配置HTTPS周围的一切（即端口，证书等）
- `Authentication` 设置委派身份验证，如“[委派身份验证和信任”中所述。](#)
- `Authorization` 按照“[委派授权”中的说明](#)设置委派授权。
- `Audit` 设置审计输出堆栈。默认情况下禁用此选项，但可以将其设置为输出审核日志文件或将审核事件发送到外部后端。
- `Features` 提供配置 alpha 和 beta 特征的特征门。
- `CoreAPI` 保存 `kubeconfig` 文件的路径以访问主 API 服务器。默认使用群集内配置。
- `Admission` 是一堆变异和验证的准入插件，可以为每个传入的 API 请求执行。这可以通过自定义代码内准入插件进行扩展，也可以针对自定义 API 服务器调整默认准入链。
- `ExtraAdmissionInitializers` 允许我们添加更多初始化者入学。初始化程序通过自定义 API 服务器实现例如线程或客户端的管道。有关自定义录取的更多信息，请参阅“[录取](#)”。
- `ProcessInfo` 保存事件对象创建的信息（即进程名称和命名空间）。我们已经 `pizza-apiserver` 为两个值设置了它。
- `Webhook` 配置 webhook 的操作方式（例如，身份验证和入场 webhook 的一般设置）。它为在集群内运行的自定义 API 服务器设置了良好的默认值。对于群集外部的 API 服务器，这将是配置它如何到达 webhook 的地方。

选项与标志相结合；也就是说，它们通常与标志处于相同的抽象级别。根据经验，选项不包含“运行”数据结构。它们在启动期间使用，然后转换为配置或服务器对象，然后运行它们。

可以通过该 `Validate()` 方法验证选项。此方法还将检查用户提供的标志值是否具有逻辑意义。

可以完成选项以设置默认值，默认值不应显示在标志的帮助文本中，但这些默认值是获取完整选项集所必需的。

通过该 `Config() (*apiserver.Config, error)` 方法将选项转换为服务器配置（“config”）。这是通过从推荐的默认配置开始，然后将选项应用于它来完成的：

```

func (o *CustomServerOptions) Config() (*apiserver.Config, error) {
    err := o.RecommendedOptions.SecureServing.MaybeDefaultWithSelfSignedCerts(
        "localhost", nil, []net.IP{net.ParseIP("127.0.0.1")},
    )
    if err != nil {
        return nil, fmt.Errorf("error creating self-signed cert: %v", err)
    }
}

```

```

    }

    [... omitted o.RecommendedOptions.ExtraAdmissionInitializers ...]

    serverConfig := genericapiserver.NewRecommendedConfig(apiserver.Codecs)
    err = o.RecommendedOptions.ApplyTo(serverConfig, apiserver.Scheme);
    if err != nil {
        return nil, err
    }

    config := &apiserver.Config{
        GenericConfig: serverConfig,
        ExtraConfig:   apiserver.ExtraConfig{},
    }
    return config, nil
}

```

这里创建的配置包含可运行的数据结构; 换句话说, 配置是运行时对象, 与对应于标志的选项形成对比。`o.RecommendedOptions.SecureServing.MaybeDefaultWithSelfSignedCerts` 如果用户未传递预生成证书的标志, 该行将创建自签名证书。

正如我们所描述的, `genericapiserver.NewRecommendedConfig` 返回默认的推荐配置, 并 `RecommendedOptions.ApplyTo` 根据标志 (和其他自定义选项) 进行更改。

`pizza-apiserver` 项目本身的配置结构只是 `RecommendedConfig` 我们的示例自定义API服务器的包装:

```

type ExtraConfig struct {
    // Place your custom config here.
}

type Config struct {
    GenericConfig *genericapiserver.RecommendedConfig
    ExtraConfig   ExtraConfig
}

// CustomServer contains state for a Kubernetes custom api server.
type CustomServer struct {
    GenericAPIServer *genericapiserver.GenericAPIServer
}

type completedConfig struct {
    GenericConfig genericapiserver.CompletedConfig
    ExtraConfig   *ExtraConfig
}

type CompletedConfig struct {
    // Embed a private pointer that cannot be instantiated outside of
    // this package.
    *completedConfig
}

```

```
}
```

如果需要更多运行自定义API服务器的状态，`ExtraConfig` 则可以放置它。

与选项结构类似，配置有一个 `Complete()` `CompletedConfig` 设置默认值的方法。因为有必要实际调用 `Complete()` 底层配置，所以通常通过引入未导出的 `completedConfig` 数据类型来强制通过类型系统实现。这里的想法是只有一个电话 `Complete()` 可以 `config` 变成一个 `completeConfig`。如果没有完成此调用，编译器将会抱怨：

```
func (cfg *Config) Complete() completedConfig {
    c := completedConfig{
        cfg.GenericConfig.Complete(),
        &cfg.ExtraConfig,
    }

    c.GenericConfig.Version = &version.Info{
        Major: "1",
        Minor: "0",
    }

    return completedConfig{&c}
}
```

最后，完成的配置可以 `CustomServer` 通过 `New()` 构造函数转换为运行时结构：

```
// New returns a new instance of CustomServer from the given config.
func (c completedConfig) New() (*CustomServer, error) {
    genericServer, err := c.GenericConfig.New(
        "pizza-apiserver",
        genericapiserver.NewEmptyDelegate(),
    )
    if err != nil {
        return nil, err
    }

    s := &CustomServer{
        GenericAPIServer: genericServer,
    }

    [ ... omitted API installation ...]

    return s, nil
}
```

请注意，我们在此有意省略了API安装部分。我们将在“[API安装](#)”中回顾这一点（即，在启动期间如何将注册表连接到自定义API服务器）。注册表实现API组的API和存储语义。我们将在“[注册表和策略](#)”中看到餐厅API组。

`CustomServer` 最终可以使用该 `Run(stopCh <-chan struct{}) error` 方法启动该对象。这是通过 `Run` 我们示例中的选项方法调用的。那就是 `CustomServerOptions.Run`：

- 创建配置
- 完成配置
- 创造 `CustomServer`
- 呼叫 `CustomServer.Run`

这是代码：

```
func (o CustomServerOptions) Run(stopCh <-chan struct{}) error {
    config, err := o.Config()
    if err != nil {
        return err
    }

    server, err := config.Complete().New()
    if err != nil {
        return err
    }

    server.GenericAPIServer.AddPostStartHook("start-pizza-apiserver-informers",
        func(context genericapiserver.PostStartHookContext) error {
            config.GenericConfig.SharedInformerFactory.Start(context.StopCh)
            o.SharedInformerFactory.Start(context.StopCh)
            return nil
        },
    )

    return server.GenericAPIServer.PrepareRun().Run(stopCh)
}
```

该 `PrepareRun()` 调用连接了OpenAPI规范，并可能执行其他API后安装操作。调用它之后，该 `Run` 方法启动实际的服务器。它会阻塞直到 `stopCh` 关闭。

这个示例还连接一个名为的启动后挂钩 `start-pizza-apiserver-informers`。顾名思义，在HTTPS服务器启动和监听后调用启动后挂钩。在这里，它启动了共享的informer工厂。

请注意，即使是自定义API服务器本身提供的资源的本地进程内信息，也可以通过HTTPS向localhost接口说明。因此，在服务器启动且HTTPS端口正在侦听之后启动它们是有意义的。

另请注意，只有在所有启动后挂钩成功完成后，`/healthz`端点才会返回成功。

随着所有小管道部件的到位，`pizza-apiserver` 项目将所有内容包装成一个 `cobra` 命令：

```
// NewCommandStartCustomServer provides a CLI handler for 'start master' command
// with a default CustomServerOptions.
func NewCommandStartCustomServer(
    defaults *CustomServerOptions,
    stopCh <-chan struct{},
```

```

) *cobra.Command {
    o := *defaults
    cmd := &cobra.Command{
        Short: "Launch a custom API server",
        Long:  "Launch a custom API server",
        RunE: func(c *cobra.Command, args []string) error {
            if err := o.Complete(); err != nil {
                return err
            }
            if err := o.Validate(); err != nil {
                return err
            }
            if err := o.Run(stopCh); err != nil {
                return err
            }
            return nil
        },
    }

    flags := cmd.Flags()
    o.RecommendedOptions.AddFlags(flags)

    return cmd
}

```

使用 `NewCommandStartCustomServer` 该 `main()` 过程的方法非常简单：

```

func main() {
    logs.InitLogs()
    defer logs.FlushLogs()

    stopCh := genericapiserver.SetupSignalHandler()
    options := server.NewCustomServerOptions(os.Stdout, os.Stderr)
    cmd := server.NewCommandStartCustomServer(options, stopCh)
    cmd.Flags().AddGoFlagSet(flag.CommandLine)
    if err := cmd.Execute(); err != nil {
        klog.Fatal(err)
    }
}

```

特别注意调用 `SetupSignalHandler`：它连接Unix信号处理。开 `SIGINT`（在终端中按Ctrl-C时触发）`SIGKILL`，停止通道关闭。停止通道将传递给正在运行的自定义API服务器，并在停止通道关闭时关闭。因此，当接收到其中一个信号时，主循环将启动关闭。在终止之前运行请求完成（默认情况下最多60秒）的意义上，此关闭是优雅的。它还确保将所有请求发送到审计后端，并且不会删除任何审计数据。毕竟，`cmd.Execute()` 将返回并且该过程将终止。

第一个开始

现在我们已经准备好了第一次启动自定义API服务器的所有内容。假设您在`~/.kube/config`中配置了一个集群，您可以将其用于委派身份验证和授权：

```
$ cd $GOPATH/src/github.com/programming-kubernetes/pizza-apiserver
$ etcd &
$ go run . --etcd-servers localhost: 2379 \
--authentication-kubeconfig~ / .kube / config \
--authorization-kubeconfig~ / .kube / config \
--kubeconfig~ / .kube / config
I0331 11: 33: 25.702320    64244plugins.go: 158 ]
加载3突变接纳控制器(小号)成功以下顺序:
  NamespaceLifecycle, MutatingAdmissionWebhook, PizzaToppings。
I0331 11: 33: 25.702344    64244plugins.go: 161 ]
加载1验证许可控制器(小号)成功以下顺序:
  ValidatingAdmissionWebhook。
I0331 11: 33: 25.714148    64244secure_serving.go: 116 ]上的安全服务[:: ]: 443
```

它将启动并开始提供通用API端点：

```
$ curl -k https://localhost: 443 / healthz
好
```

我们也可以列出发现端点，但结果还不是很令人满意 - 我们还没有创建API，所以发现是空的：

```
$ curl -k https://localhost: 443 / apis
{
  "kind": "APIGroupList",
  "groups": []
}
```

我们来看一个更高层次：

- 我们已经使用推荐的选项和配置启动了自定义API服务器。
- 我们有一个标准的处理程序链，包括委托身份验证，委派授权和审计。
- 我们有一台HTTPS服务器正在运行并提供对通用端点的请求：`/logs`, `/metrics`, `/version`, `/healthz`和`/apis`。

[图8-3](#)显示了10,000英尺的距离。

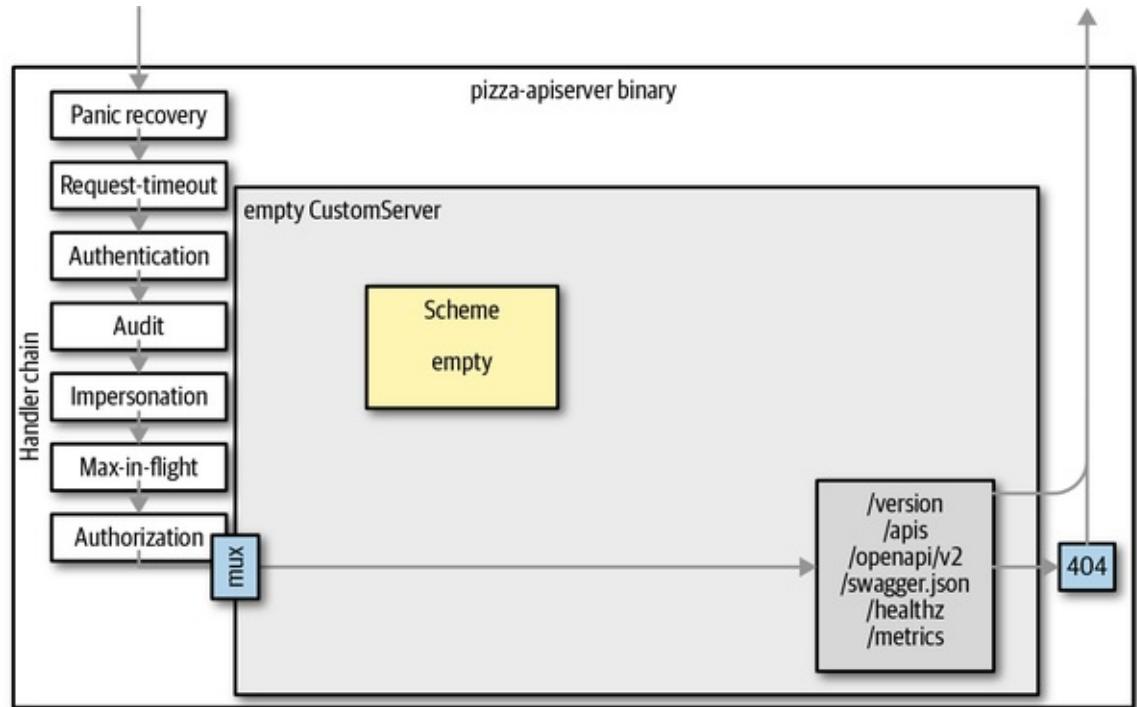


图8-3。没有API的自定义API服务器

内部类型和转换

现在我们已经设置了一个运行的自定义API服务器，是时候实际实现API了。在此之前，我们必须了解API版本以及如何在API服务器内处理它们。

每个API服务器都提供许多资源和版本（参见图2-3）。有些资源有多个版本。为了使资源的多个版本成为可能，API服务器 转换版本。

为避免版本之间必要转换的二次增长，API服务器使用 实现实际API逻辑时的内部版本。内部版本也经常被调用 集线器版本，因为它是一种集合，每个其他版本都转换为和从中转换（见图8-4）。内部API逻辑仅针对该集线器版本实现一次。

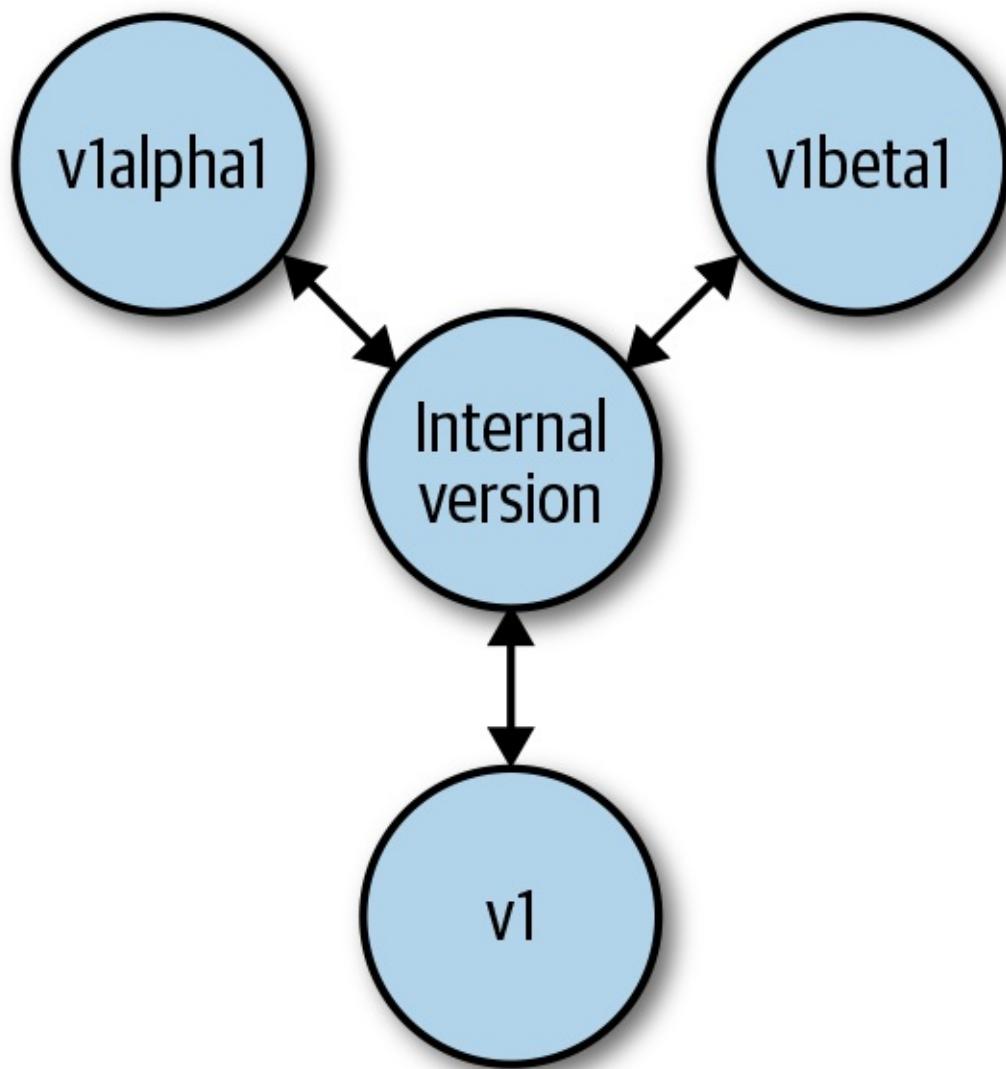


图8-4。从集线器版本转换到集线器版本

[图8-5](#)显示了API服务器如何在API请求的生命周期中使用内部版本：

- 用户使用特定版本（例如，`v1`）发送请求。
- API服务器解码有效负载并将其转换为内部版本。
- API服务器通过许可和验证传递内部版本。
- API逻辑是为注册表中的内部版本实现的。
- `etcd` 读取和写入版本化对象（例如，`v2` - 存储版本）；也就是说，它从内部版本转换为内部版本。
- 最后，在这种情况下，结果将转换为请求版本 `v1`。

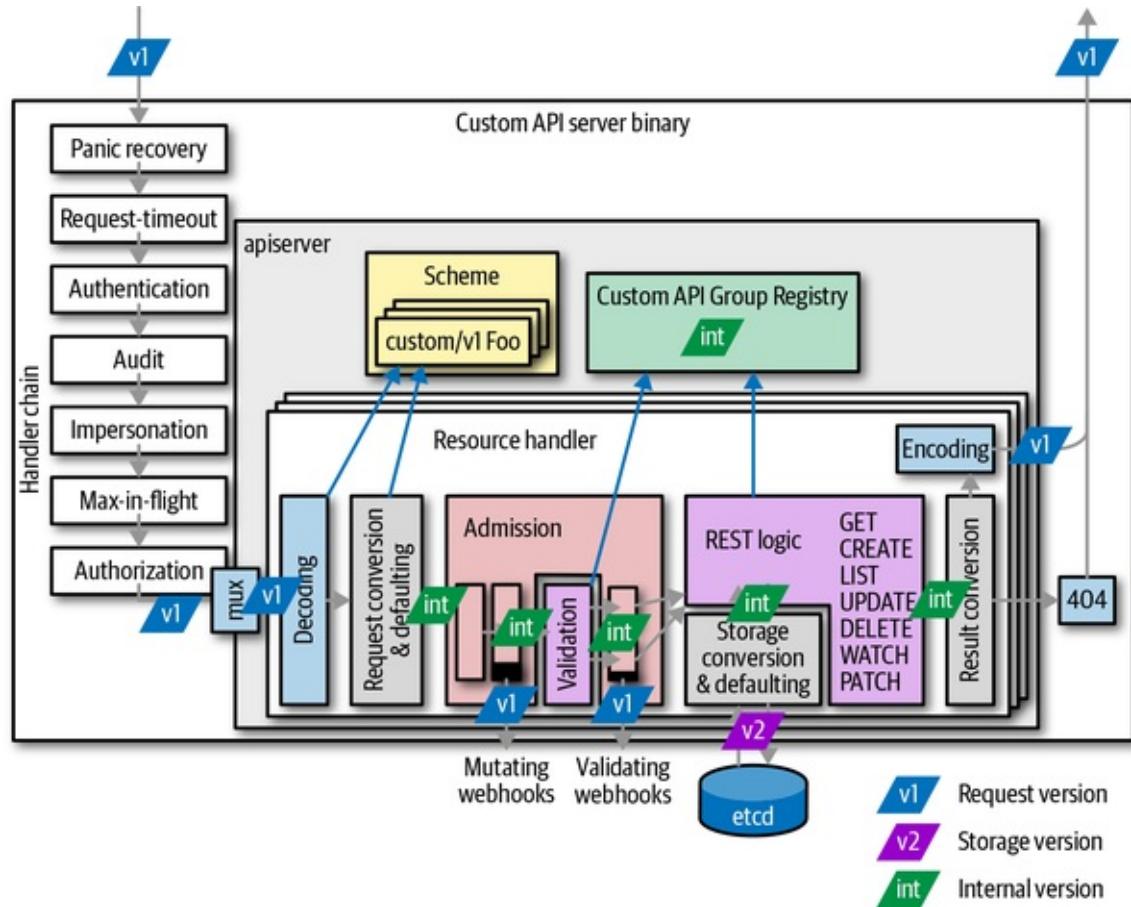


图8-5。在请求的生命周期中转换API对象

在内部集线器版本和之间的每个边缘外部版本，进行转换。在图8-6中，您可以计算每个请求处理程序的转换次数。在写入操作（如创建和更新）中，至少完成四次转换，如果在群集中部署了许可 webhook，则会进行更多转换。如您所见，转换是每个API实现中的关键操作。

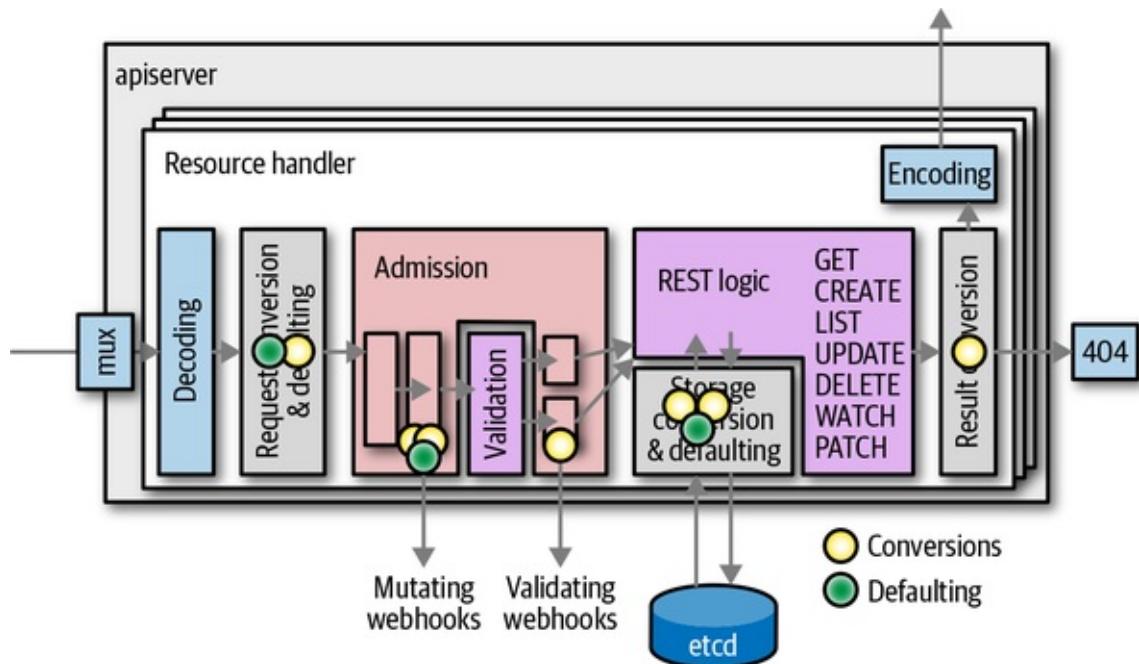


图8-6。在请求的生命周期中进行转换和默认

在除转换外，[图8-6](#)还显示了何时发生默认情况。默认是填写未指定字段值的过程。默认与转换高度耦合，并且当来自用户的请求，来自 etcd 或来自接纳 webhook 时，始终在外部版本上完成，但从从集线器转换到外部版本时永远不会。

警告

转变对API服务器机制至关重要。所有转换（来回）在圆形转换的意义上必须是正确的也是至关重要的。Roundtrippable 意味着我们可以在版本图中来回转换（[图8-4](#)），从随机值开始，我们永远不会丢失任何信息；也就是说，转换是双向的，或者是一对一的。例如，我们必须能够从随机（但有效）v1 对象转到内部集线器类型，然后 v1alpha1 返回到内部集线器类型，然后返回到 v1。生成的对象必须等同于原始对象。

制作可环绕的类型通常需要经过深思熟虑；它几乎总是驱动新版本的API设计，并且还影响旧类型的扩展，以便存储新版本携带的信息。

简而言之：正确地进行往返是很困难的。请参阅[“往返测试”](#)以了解如何有效地测试往返。

在API服务器的生命周期中，默认逻辑可能会发生变化。想象一下，您为类型添加了一个新字段。用户可能将旧对象存储在磁盘上，或者 etcd 可能具有旧对象。如果该新字段具有默认值，则在将旧的存储对象发送到API服务器时或者当用户从中检索其中一个旧对象时设置此字段值 etcd 。看起来新字段永远存在，而实际上API服务器中的默认进程在处理请求期间设置字段值。

编写API类型

如我们已经看到，要向自定义API服务器添加API，我们必须编写内部集线器版本类型和外部版本类型，并在它们之间进行转换。这就是我们现在要看的[披萨示例项目](#)。

API类型在传统上是为PKG的`/apis/group-name`包与该项目的PKG的`/apis/group-name/types.go`内部类型和PKG的`/apis/group-name/version/types.go`用于外部的版本）。因此，对于我们的示例，`pkg/apis/restaurant`, `pkg/apis/restaurant/v1alpha1/types.go`和`pkg/apis/restaurant/v1beta1/types.go`。

转换将在`pkg/apis/group-name/version/zz_generated.conversion.go`（用于 `conversion-gen` 输出）和`pkg/apis/group-name/version/conversion.go`中创建，用于开发人员编写的自定义转换。

以类似的方式，将为`pkg/apis//zz_generated.defaults.go`和`pkg/apis//defaults.go`上的 `defaulter-gen` 输出创建默认代码，以获取开发人员编写的自定义默认代码。在我们的例子中，我们有`pkg/apis/restaurant/v1alpha1/defaults.go`和`pkg/apis/restaurant/v1beta1/defaults.go`。`group-nameversion**group-nameversion`

我们将详细介绍转换和默认“[转换](#)”和“[默认](#)”。

除了转换和默认之外，我们已经在[“剖析类型”](#)中看到了CustomResourceDefinitions的大部分过程。我们的自定义API服务器中的外部版本的本机类型的定义方式完全相同。

另外，对于内部类型，集线器类型，我们有`pkg/apis/group-name/types.go`。主要区别在于后者`SchemeGroupVersion`在`register.go`文件中引用`runtime.APIVersionInternal`（这是一个快捷方式“`__internal`”）。

```
// SchemeGroupVersion is group version used to register these objects
var SchemeGroupVersion = schema.GroupVersion{Group: GroupName, Version:
    runtime.APIVersionInternal}
```

和外部类型文件之间的另一个区别是缺少JSON和protobuf标签。`pkg/apis/*group-name*/types.go`

小费

某些生成器使用JSON标记来检测`types.go`文件是用于外部版本还是内部版本。因此，在复制和粘贴外部类型时，请始终删除这些标记，以便创建或更新内部类型。

最后但并非最不重要的是，有一个帮助程序可以将API组的所有版本安装到方案中。传统上，此助手位于`pkg/apis/group-name/install/install.go`中。对于我们的自定义API服务器`pkg/apis/restaurant/install/install.go`，它看起来很简单：

```
// Install registers the API group and adds types to a scheme
func Install(scheme *runtime.Scheme) {
    utilruntime.Must(restaurant.AddToScheme(scheme))
    utilruntime.Must(v1beta1.AddToScheme(scheme))
    utilruntime.Must(v1alpha1.AddToScheme(scheme))
    utilruntime.Must(scheme.SetVersionPriority(
        v1beta1.SchemeGroupVersion,
        v1alpha1.SchemeGroupVersion,
    ))
}
```

因为我们有多个版本，所以必须定义优先级。此订单将用于确定资源的默认存储版本。它曾经在内部客户端（返回内部版本对象的客户端）中的版本选择中发挥作用；请参阅注释[“过去的版本化客户端和内部客户端”](#)。但是内部客户已被弃用并且正在消失。即使API服务器内的代码将来也会使用外部版本客户端。

转换

转变获取一个版本中的对象并将其转换为另一个版本中的对象。转换是通过转换函数实现的，其中一些是手工编写的（按惯例放入`pkg/apis/group-name//versionconversion.go`），另一些是自动生成的`conversion-gen`（按惯例放入`pkg/apis/group-name/version/zz_generated.conversion.go`）。

使用该方法通过方案（参见[“Scheme”](#)）启动转换`Convert()`，传递源对象`in`和目标对象`out`：

```
func (s *Scheme) Convert(in, out interface{}, context interface{}) error
```

将`context`被描述为如下：

```
// ...an optional field that callers may use to pass info to conversion functions.
```

它仅在非常特殊的情况下使用，通常是 `nil`。在本章的后面，我们将介绍转换函数范围，它允许我们从转换函数中访问此上下文。

为了进行实际转换，该方案了解所有Golang API类型，它们的类型`GroupVersionKinds`，以及`GroupVersionKinds`之间的转换函数。为此，`conversion-gen` 寄存器通过本地方案构建器生成转换函数。在我们的示例自定义API服务器中，`zz_generated.conversion.go`文件的开头如下：

```
func init() {
    localSchemeBuilder.Register(RegisterConversions)
}

// RegisterConversions adds conversion functions to the given scheme.
// Public to allow building arbitrary schemes.
func RegisterConversions(s *runtime.Scheme) error {
    if err := s.AddGeneratedConversionFunc(
        (*Topping)(nil),
        (*restaurant.Topping)(nil),
        func(a, b interface{}, scope conversion.Scope) error {
            return Convert_v1alpha1_Topping_To_restaurant_Topping(
                a.(*Topping),
                b.(*restaurant.Topping),
                scope,
            )
        },
    ); err != nil {
        return err
    }
    ...
    return nil
}

...
```

`Convert_v1alpha1_Topping_To_restaurant_Topping()` 生成该功能。它需要一个 `v1alpha1` 对象并将其转换为内部类型。

注意

前面的复杂类型转换将类型转换函数转换为统一类型 `func(a, b interface{}, scope conversion.Scope) error`。该方案使用后者类型，因为它可以在不使用反射的情况下调用它们。由于许多必要的分配，反思很慢。

在手写转换`conversion.go`在一定意义生成过程中优先考虑 `conversion-gen` 跳过一代的类型，如果它与包找到了手写功能`Convert source-package-basename_KindTo target-package-basename_Kind`转换功能命名模式。例如：

```

func Convert_v1alpha1_PizzaSpec_To_restaurant_PizzaSpec(
    in *PizzaSpec,
    out *restaurant.PizzaSpec,
    s conversion.Scope,
) error {
    ...
    return nil
}

```

在最简单的情况下，转换函数只是将值从源复制到目标对象。但是对于将 v1alpha1 披萨规范转换为内部类型的前一个示例，简单复制是不够的。我们必须调整不同的结构，实际上如下所示：

```

func Convert_v1alpha1_PizzaSpec_To_restaurant_PizzaSpec(
    in *PizzaSpec,
    out *restaurant.PizzaSpec,
    s conversion.Scope,
) error {
    idx := map[string]int{}
    for _, top := range in.Toppings {
        if i, duplicate := idx[top]; duplicate {
            out.Toppings[i].Quantity++
            continue
        }
        idx[top] = len(out.Toppings)
        out.Toppings = append(out.Toppings, restaurant.PizzaTopping{
            Name: top,
            Quantity: 1,
        })
    }

    return nil
}

```

显然，没有代码生成可以如此聪明，以至于可以预见用户在定义这些不同类型时的意图。

请注意，在转换期间，源对象绝不能变异。但这是完全正常的，并且通常出于性能原因，强烈建议在类型匹配时重用目标对象中的源的数据结构。

这非常重要，我们在警告中重申它，因为它不仅对转换的实现有影响，而且对转换的调用者和转换输出的消费者也有影响。

警告

转换函数不得改变源对象，但允许输出与源共享数据结构。这意味着转换输出的使用者必须确保在原始对象不得变异的情况下不要改变对象。

例如，假设您 `pod *core.Pod` 在内部版本中有一个，并将其转换为 `v1 as podv1 *corev1.Pod`，并对结果进行变更 `podv1`。这也可能会改变原作 `pod`。如果 `pod` 来自 `informer`，这是非常危险的，因为 `informers` 有一个共享缓存，并且变异 `pod` 使得缓存不一致。

因此，请注意转换的这种属性，并在必要时进行深层复制，以避免意外和潜在危险的突变。

虽然这种数据结构的共享会带来一些风险，但它也可以避免在许多情况下进行不必要的分配。生成的代码到目前为止，生成器比较源和目标结构，并使用 Go 的 `unsafe` 包通过简单的类型转换将指针转换为相同内存布局的结构。因为 `v1beta1` 我们示例中的披萨的内部类型和类型具有相同的内存布局，所以我们得到：

```
func autoConvert_restaurant_PizzaSpec_To_v1beta1_PizzaSpec(
    in *restaurant.PizzaSpec,
    out *PizzaSpec,
    s conversion.Scope,
) error {
    out.Toppings = *(*[]PizzaTopping)(unsafe.Pointer(&in.Toppings))
    return nil
}
```

在机器语言级别，这是一个 NOOP，因此它可以尽可能快。它避免在这种情况下分配切片并逐项复制 `in` 到 `out`。

最后但并非最不重要的是，关于转换函数的第三个参数的一些说法：转换范围 `conversion.Scope`。

转换范围提供对许多转换元级别值的访问。例如，它允许我们 `context` 通过以下方式访问传递给方案 `Convert(in, out interface{}, context interface{}) error` 方法的值：

```
s.Meta().Context
```

它还允许我们通过 `s.Convert` 或不考虑注册的转换函数来调用子类型的方案转换 `s.DefaultConvert`。

但是，在大多数转换情况下，根本不需要使用范围。为简单起见，您可以忽略它的存在，直到您遇到需要比源和目标对象更多的上下文的棘手情况。

违约

违约是 API 请求生命周期中的步骤，它为传入对象（来自客户端或来自 `etcd`）中的省略字段设置默认值。例如，`pod` 有一个 `restartPolicy` 字段。如果用户未指定它，则默认值为 `Always`。

想象一下，我们在 2014 年左右使用了一个非常古老的 Kubernetes 版本。该领域 `restartPolicy` 刚刚在当时的最新版本中引入了该系统。升级群集后，如果 `etcd` 没有该 `restartPolicy` 字段，则会有一个窗格。一 `kubectl get pod` 会从中读取旧 `pod`，`etcd` 默认代码将添加默认值 `Always`。从用户的角度来看，神奇的老吊舱突然有了新的 `restartPolicy` 领域。

请参阅图8-6，了解Kubernetes请求管道中今天发生的默认操作。请注意，默认仅针对外部类型而非内部类型。

现在让我们看一下默认的代码。默认是由k8s.io/apiserver代码通过该方案启动的，类似于转换。因此，我们必须将默认函数注册到我们的自定义类型的方案中。

同样，与转换类似，大多数默认代码只是使用 `defaulter-gen` 二进制文件生成的。它遍历API类型并在 `pkg/apis/group-name/version/zz_generated.defaults.go` 中创建默认函数。除了为子结构调用默认函数之外，默认情况下代码不会执行任何操作。

您可以通过遵循默认函数命名模式来定义自己的默认逻辑：`SetDefaults*Kind*`

```
func SetDefaultsKind(obj *Type) {
    ...
}
```

此外，与转换不同，我们必须手动调用本地方案构建器上生成的函数的注册。遗憾的是，这不是自动完成的：

```
func init() {
    localSchemeBuilder.Register(RegisterDefaults)
}
```

这里 `RegisterDefaults` 是在包 `pkg/apis/group-name/version/zz_generated.defaults.go` 中生成的。

对于默认代码，了解用户何时设置字段以及何时不设置字段至关重要。在许多情况下，这并不是那么清楚。

Golang对于每种类型都没有值，如果在传递的JSON或protobuf中找不到字段，则设置它们。想象一下 `true` 布尔字段的默认值 `foo`。零值是 `false`。不幸的是，不清楚是否 `false` 由于用户的输入而设置，或者因为 `false` 只是布尔值的零值。

为了避免这种情况，通常必须在Golang API类型中使用指针类型（例如，`*bool` 在前面的例子中）。用户提供的 `false` 将导致 `nil` 指向 `false` 值的非布尔指针，并且用户提供的 `true` 将导致非 `nil` 布尔指针和 `true` 值。没有提供的字段导致 `nil`。这可以在默认代码中检测到：

```
func SetDefaultsKind(obj *Type) {
    if obj.Foo == nil {
        x := true
        obj.Foo = &x
    }
}
```

这给出了所需的语义：“`foo`默认为`true`”。

小费

这种使用指针的技巧适用于像字符串这样的基本类型。对于地图和数组，如果不识别 `nil` 地图/数组和空地图/数组，通常很难达到可循环性。因此，Kubernetes中用于地图和数组的大多数默认程序在两种情况下都应用默认值，即解决编码和解码错误。

往返测试

获得转换对，很难。往返测试是一项必不可少的工具，可以在随机测试中自动检查转换是否按计划进行，并且在转换为所有已知组版本时不会丢失数据。

往返测试通常与 `install.go` 文件一起放置（例如，放入 `pkg/apis/restaurant/install/roundtrip_test.go`），然后从 API Machinery 调用往返测试函数：

```
import (
    ...
    "k8s.io/apimachinery/pkg/api/apitest/roundtrip"
    restaurantfuzzer "github.com/programming-kubernetes/pizza-apiserver/pkg/apis/restaurant/fuzzer"
)

func TestRoundTripTypes(t *testing.T) {
    roundtrip.RoundTripTestForAPIGroup(t, Install, restaurantfuzzer.Funcs)
}
```

在内部，`RoundTripTestForAPIGroup` 调用使用 `Install` 函数将 API 组安装到临时方案中。然后，它使用给定的模糊器在内部版本中创建随机对象，然后将它们转换为某个外部版本并返回到内部版本。生成的对象必须与原始对象等效。所有外部版本的测试都进行了数百次或数千次。

一个模糊器为内部类型及其子类型返回一组随机函数的函数。在我们的示例中，模糊器放在包 `pkg/apis/restaurant/fuzzer/fuzzer.go` 中，并包含 spec 结构的随机函数：

```
// Funcs returns the fuzzing functions for the restaurant api group.
var Funcs = func(codecs runtime.CodecFactory) []interface{} {
    return []interface{}{
        func(s *restaurant.PizzaSpec, c fuzz.Continue) {
            c.FuzzNoCustom(s) // fuzz first without calling this function again

            // avoid empty Toppings because that is defaulted
            if len(s.Toppings) == 0 {
                s.Toppings = []restaurant.PizzaTopping{
                    {"salami", 1},
                    {"mozzarella", 1},
                    {"tomato", 1},
                }
            }

            seen := map[string]bool{}
            for i := range s.Toppings {
                // make quantity strictly positive and of reasonable size
            }
        },
    }
}
```

```
s.Toppings[i].Quantity = 1 + c.Intn(10)

        // remove duplicates
        for {
            if !seen[s.Toppings[i].Name] {
                break
            }
            s.Toppings[i].Name = c.RandString()
        }
        seen[s.Toppings[i].Name] = true
    }
},
}
```

如果没有给出随机函数功能，底层库github.com/google/gofuzz通常会尝试通过设置基类型的随机值并递归地潜入指针，结构，映射和切片来模糊对象，最终调用自定义随机函数它们是由开发人员提供的。

当为其中一种类型编写随机函数函数时，`c.FuzzNoCustom(s)` 首先调用是很方便的。它使给定对象随机化，`s` 并为子结构调用自定义函数，但不为 `s` 自身调用。然后，开发人员可以限制和修复随机值以使对象有效。

警告

为了覆盖尽可能多的有效对象，使模糊器尽可能通用非常重要。如果模糊限制器过于严格，则测试覆盖率会很差。在许多情况下，在Kubernetes的开发过程中，没有发现回归，因为现场的模糊器不好。

另一方面，模糊器只需要考虑验证的对象，并且是外部版本中可定义的实际对象的投影。通常，您必须以 `c.FuzzNoCustom(s)` 随机对象变为有效的方式限制设置的随机值。例如，如果验证将拒绝任意字符串，则持有URL的字符串不必为任意值进行往返。

我们前面的 PizzaSpec 示例首先通过以下方式调用 `c.FuzzNoCustom(s)` 并修复对象：

- 违反 `nil` 配料的情况
 - 为每个顶部设置合理的数量（没有它，转换 `v1alpha1` 将在复杂性中爆炸，将大量数量引入字符串列表）
 - 标准化顶部名称，因为我们知道披萨规格中的重复顶部将永远不会往返（对于内部类型，请注意 `v1alpha1` 类型有重复）

验证

传入的对象在被反序列化，默认并转换为内部版本后不久即被验证。[图8-5](#)显示了之前如何进行验证：改
变录入插件和验证许可插件，早在实际创建或更新逻辑执行之前。

这意味着验证必须仅针对内部版本实施一次，而不是针对所有外部版本实施。这样做的好处是它显然可以节省实现工作并确保版本之间的一致性。另一方面，这意味着验证错误不涉及外部版本。实际上可以通过Kubernetes资源观察到这一点，但在实践中它并没有什么大不了的。

在本节中，我们将介绍验证函数的实现。自定义API服务器的连接 - 即从配置通用注册表的策略调用验证 - 将在下一节中介绍。换句话说，[图8-5](#)略微误导，有利于视觉简洁性。

现在，它应该足以查看策略内部验证的切入点：

```
func (pizzaStrategy) Validate(
    ctx context.Context, obj runtime.Object,
) field.ErrorList {
    pizza := obj.(*restaurant.Pizza)
    return validation.ValidatePizza(pizza)
}
```

这将调用API组验证包中的验证功能。`Validate*Kind*(obj **Kind*)`
`field.ErrorList` `pkg/apis/*group*/validation*`

验证函数返回错误列表。它们通常以相同的样式编写，将返回值附加到错误列表，同时递归潜入类型，每个结构一个验证函数：

```
// ValidatePizza validates a Pizza.
func ValidatePizza(f *restaurant.Pizza) field.ErrorList {
    allErrs := field.ErrorList{}

    errs := ValidatePizzaSpec(&f.Spec, field.NewPath("spec"))
    allErrs = append(allErrs, errs...)

    return allErrs
}

// ValidatePizzaSpec validates a PizzaSpec.
func ValidatePizzaSpec(
    s *restaurant.PizzaSpec,
    fldPath *field.Path,
) field.ErrorList {
    allErrs := field.ErrorList{}

    prevNames := map[string]bool{}
    for i := range s.Toppings {
        if s.Toppings[i].Quantity <= 0 {
            allErrs = append(allErrs, field.Invalid(
                fldPath.Child("toppings").Index(i).Child("quantity"),
                s.Toppings[i].Quantity,
                "cannot be negative or zero",
            ))
        }
        if len(s.Toppings[i].Name) == 0 {
            allErrs = append(allErrs, field.Invalid(
                fldPath.Child("toppings").Index(i).Child("name"),
                s.Toppings[i].Name,
                "cannot be empty",
            ))
        }
    }
}
```

```

        fldPath.Child("toppings").Index(i).Child("name"),
        s.Toppings[i].Name,
        "cannot be empty",
    ))
} else {
    if prevNames[s.Toppings[i].Name] {
        allErrs = append(allErrs, field.Invalid(
            fldPath.Child("toppings").Index(i).Child("name"),
            s.Toppings[i].Name,
            "must be unique",
        ))
    }
    prevNames[s.Toppings[i].Name] = true
}
}

return allErrs
}

```

请注意如何使用 `Child` 和 `Index` 调用字段路径。字段路径是JSON路径，在出错时打印。

通常还有一组额外的验证函数，这些函数在更新时稍有不同（前面的集用于创建）。在我们的示例API服务器中，它可能如下所示：

```

func (pizzaStrategy) ValidateUpdate(
    ctx context.Context,
    obj, old runtime.Object,
) field.ErrorList {
    objPizza := obj.(*restaurant.Pizza)
    oldPizza := old.(*restaurant.Pizza)
    return validation.ValidatePizzaUpdate(objPizza, oldPizza)
}

```

这可用于验证是否未更改只读字段。通常，更新验证也会调用正常的验证函数，并且只会添加与更新相关的检查。

注意

验证是在创建时限制对象名称的正确位置 - 例如，仅限单个字，或不包含任何非字母数字字符。

实际上，任何 `ObjectMeta` 字段在技术上都可以以自定义方式受到限制，尽管这对于许多字段来说并不可取，因为它可能会破坏核心API机制行为。许多资源限制名称，例如，名称将显示在其他系统或需要特殊格式名称的其他上下文中。

但即使 `ObjectMeta` 自定义API服务器中存在特殊验证，通用注册表也会在自定义验证通过后对任何情况下的通用规则进行验证。这允许我们首先从自定义代码返回更具体的错误消息。

注册和战略

到目前为止，我们已经了解了如何定义和验证API类型。下一步是为这些API类型实现REST逻辑。图8-7显示了注册表作为API组实现的核心部分。[k8s.io/apiserver](#)中的通用REST请求处理程序代码调用注册表。

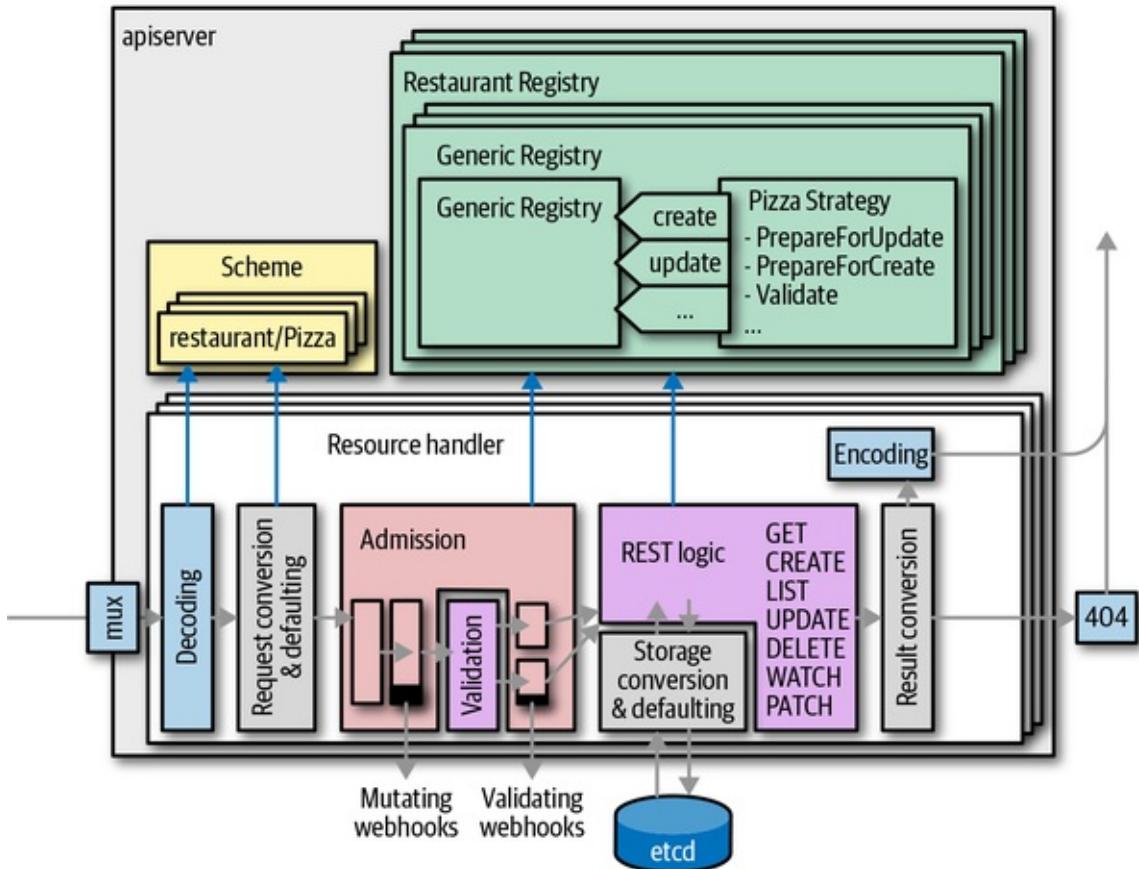


图8-7。资源存储和通用注册表

通用注册表

该REST逻辑通常由所谓的通用注册表实现。顾名思义，它是[k8s.io/apiserver/pkg/registry/rest](#)包中注册表接口的通用实现。

通用注册表实现“普通”资源的默认REST行为。几乎所有Kubernetes资源都使用此实现。只有少数，特别是那些没有持久化对象（例如，[SubjectAccessReview](#) 参见“[委托授权](#)”），具有自定义实现。

在[k8s.io/apiserver/pkg/registry/rest/rest.go](#)中，您会发现许多接口，松散地对应于HTTP谓词和某些API功能。如果接口由注册表实现，则API端点代码将提供某些REST功能。由于通用注册表实现了大多数[k8s.io/apiserver/pkg/registry/rest](#)接口，因此使用它的资源将支持所有默认的Kubernetes HTTP谓词（请参阅“[API服务器的HTTP接口](#)”）。以下是使用Kubernetes源代码中的GoDoc描述实现的接口列表：

- `CollectionDeleter`

可以删除RESTful资源集合的对象

- `Creator`

可以创建RESTful对象实例的对象

- `CreaterUpdater`

必须支持创建和更新操作的存储对象

- `Exporter`

一个知道如何剥离RESTful资源以进行导出的对象

- `Getter`

可以检索命名的RESTful资源的对象

- `GracefulDeleter`

一个知道如何传递删除选项以允许延迟删除RESTful对象的对象

- `Lister`

可以检索与提供的字段和标签条件匹配的资源的对象

- `Patcher`

支持get和update的存储对象

- `Scoper`

必须指定的对象，并指示资源的范围

- `Updater`

可以更新RESTful对象实例的对象

- `Watcher`

应由所有希望提供通过 `Watch API` 监视更改的存储对象实现的对象

我们来看看其中一个接口 `Creater`：

```
// Creater is an object that can create an instance of a RESTful object.
type Creater interface {
    // New returns an empty object that can be used with Create after request
    // data has been put into it.
    // This object must be a pointer type for use with Codec.DecodeInto([]byte,
    // runtime.Object)
    New() runtime.Object

    // Create creates a new version of a resource.
    Create(
        ctx context.Context,
        obj runtime.Object,
        createValidation ValidateObjectFunc,
        options *metav1.CreateOptions,
    ) (runtime.Object, error)
}
```

```
}
```

实现此接口的注册表将能够创建对象。与此相反 `NamedCreator`，新对象的名称来自 `ObjectMeta.Name` 或通过生成 `ObjectMeta.GenerateName`。如果注册表实现 `NamedCreator`，则名称也可以通过HTTP路径传递。

重要的是要了解实现的接口确定在将API安装到自定义API服务器时创建的API端点将支持哪些谓词。有关如何在代码中完成此操作，请参阅[“API安装”](#)。

战略

该通用注册表可以使用称为策略的对象在一定程度上进行自定义。正如我们在[“验证”中](#)看到的那样，该策略为验证等功能提供了回调。

该策略使用其GoDoc描述实现此处列出的REST策略接口（有关其定义，请参阅k8s.io/apiserver/pkg/registry/rest）：

- `RESTCreateStrategy`

定义最小验证，接受的输入和名称生成行为，以创建遵循Kubernetes API约定的对象。

- `RESTDeleteStrategy`

定义遵循Kubernetes API约定的对象的删除行为。

- `RESTGracefulDeleteStrategy`

必须由支持正常删除的注册表实现。

- `GarbageCollectionDeleteStrategy`

必须由默认情况下想要孤立依赖的注册表实现。

- `RESTExportStrategy`

定义如何导出Kubernetes对象。

- `RESTUpdateStrategy`

定义更新遵循Kubernetes API约定的对象的最小验证，接受的输入和名称生成行为。

让我们再看一下创作案例的策略：

```
type RESTCreateStrategy interface {
    runtime.ObjectTyper
    // The name generator is used when the standard GenerateName field is set.
    // The NameGenerator will be invoked prior to validation.
    names.NameGenerator

    // NamespaceScoped returns true if the object must be within a namespace.
    NamespaceScoped() bool
    // PrepareForCreate is invoked on create before validation to normalize
```

```

    // the object. For example: remove fields that are not to be persisted,
    // sort order-insensitive list fields, etc. This should not remove fields
    // whose presence would be considered a validation error.
    //
    // Often implemented as a type check and an initialization or clearing of
    // status. Clear the status because status changes are internal. External
    // callers of an api (users) should not be setting an initial status on
    // newly created objects.
    PrepareForCreate(ctx context.Context, obj runtime.Object)
    // Validate returns an ErrorList with validation errors or nil. Validate
    // is invoked after default fields in the object have been filled in
    // before the object is persisted. This method should not mutate the
    // object.
    Validate(ctx context.Context, obj runtime.Object) field.ErrorList
    // Canonicalize allows an object to be mutated into a canonical form. This
    // ensures that code that operates on these objects can rely on the common
    // form for things like comparison. Canonicalize is invoked after
    // validation has succeeded but before the object has been persisted.
    // This method may mutate the object. Often implemented as a type check or
    // empty method.
    Canonicalize(obj runtime.Object)
}

```

该嵌入 `ObjectTyper` 识别对象; 也就是说, 它检查注册表是否支持请求中的对象。这对于创建正确类型的对象很重要 (例如, 通过“foo”资源, 只应创建“Foo”资源)。

该 `NameGenerator` 显然从产生的名称 `objectMeta.GenerateName` 领域。

通过 `NamespaceScoped` 该策略可以返回任何支持集群范围或命名空间资源 `false` 或 `true`。

`PrepareForCreate` 在验证之前使用传入对象调用该方法。

`Validate` 我们之前在“[验证](#)”中看到的方法: 它是验证函数的入口点。

最后, 该 `Canonicalize` 方法进行归一化 (例如, 切片的分类)。

将策略连接到通用注册表

策略对象插入通用注册表实例。以下是[GitHub上](#)我们的自定义API服务器的REST存储构造函数:

```

// NewREST returns a RESTStorage object that will work against API services.
func NewREST(
    scheme *runtime.Scheme,
    optsGetter generic.RESTOptionsGetter,
) (*registry.REST, error) {
    strategy := NewStrategy(scheme)

    store := &genericregistry.Store{
        NewFunc:      func() runtime.Object { return &restaurant.Pizza{} },
        NewListFunc:   func() runtime.Object { return &restaurant.PizzaList{} },
    }
}

```

```

        PredicateFunc: MatchPizza,

        DefaultQualifiedResource: restaurant.Resource("pizzas"),

        CreateStrategy: strategy,
        UpdateStrategy: strategy,
        DeleteStrategy: strategy,
    }
    options := &generic.StoreOptions{
        RESTOptions: optsGetter,
        AttrFunc: GetAttrs,
    }
    if err := store.CompleteWithOptions(options); err != nil {
        return nil, err
    }
    return &registry.REST{store}, nil
}

```

它实例化通用注册表对象 `genericregistry.Store` 并设置几个字段。其中许多字段都是可选字段，`store.CompleteWithOptions` 如果开发人员未设置它们，则会默认显示它们。

您可以看到自定义的策略是首先通过实例化的 `NewStrategy` 构造函数，然后插入到注册表中 `create`，`update` 和 `delete` 运营商。

此外，`NewFunc` 设置为创建新对象实例，并将该 `NewListFunc` 字段设置为创建新对象列表。将 `PredicateFunc` 选择器（可以传递给列表请求）转换为谓词函数，过滤运行时对象。

返回的对象是一个REST注册表，只是我们在通用注册表对象周围的示例项目中的一个简单包装器，以使类型成为我们自己的类型：

```

type REST struct {
    *genericregistry.Store
}

```

有了这个，我们就可以实例化我们的API并将其连接到自定义API服务器。在下一节中，我们将了解如何从中创建HTTP处理程序。

API安装

激活 API服务器中的API，需要两个步骤：

1. 必须将API版本安装到API类型中（以及转换和默认功能）服务器方案。
2. 必须将API版本安装到服务器HTTP多路复用器（mux）中。

第一步通常使用 `init` API服务器引导中集中的某个功能来完成。这是在我们的示例自定义API服务器中的 `pkg/apiserver/apiserver.go` 中完成的，其中定义了 `serverConfig` 和 `CustomServer` 对象（请参阅“[选项和配置模式和启动管道](#)”）：

```

import (
    ...
    "k8s.io/apimachinery/pkg/runtime"
    "k8s.io/apimachinery/pkg/runtime/serializer"

    "github.com/programming-kubernetes/pizza-apiserver/pkg/apis/restaurant/install"
)

var (
    Scheme = runtime.NewScheme()
    Codecs = serializer.NewCodecFactory(Scheme)
)

```

然后，对于应该提供的每个API组，我们调用该 `Install()` 函数：

```

func init() {
    install.Install(Scheme)
}

```

由于技术原因，我们还必须在计划中添加一些与发现相关的类型（这可能会在未来版本的 `k8s.io/apiserver` 中消失）：

```

func init() {
    // we need to add the options to empty v1
    // TODO: fix the server code to avoid this
    metav1.AddToGroupVersion(Scheme, schema.GroupVersion{Version: "v1"})
    // TODO: keep the generic API server from wanting this
    unversioned := schema.GroupVersion{Group: "", Version: "v1"}
    Scheme.AddUnversionedTypes(unversioned,
        &metav1.Status{},
        &metav1.APIVersions{},
        &metav1.APIGroupList{},
        &metav1.APIGroup{},
        &metav1.APIResourceList{},
    )
}

```

有了这个，我们在全局方案中注册了我们的API类型，包括转换和默认函数。换句话说，[图8-3](#)的空方案现在知道关于我们类型的所有内容。

第二步是将API组添加到HTTP多路复用器。嵌入到我们的 `CustomServer` struct 中的通用API服务器代码提供了该 `InstallAPIGroup(apiGroupInfo *APIGroupInfo) error` 方法，该方法为API组设置整个请求管道。

我们唯一要做的就是提供一个正确填充的 `APIGroupInfo` 结构。我们 `New(*CustomServer, error)` 在 `CompletedConfig` 类型的构造函数中执行此操作：

```

// New returns a new instance of CustomServer from the given config.
func (c *GenericConfig) New() (*CustomServer, error) {
    genericServer, err := c.GenericConfig.New("pizza-apiserver",
        genericapiserver.NewEmptyDelegate())
    if err != nil {
        return nil, err
    }

    s := &CustomServer{
        GenericAPIServer: genericServer,
    }

    apiGroupInfo := genericapiserver.NewDefaultAPIGroupInfo(restaurant.GroupName,
        Scheme, metav1.ParameterCodec, Codecs)

    v1alpha1storage := map[string]rest.Storage{}

    pizzaRest := pizzastorage.NewREST(Scheme, c.GenericConfig.RESTOptionsGetter)
    v1alpha1storage["pizzas"] = customregistry.RESTInPeace(pizzaRest)

    toppingRest := toppingstorage.NewREST(
        Scheme, c.GenericConfig.RESTOptionsGetter,
    )
    v1alpha1storage["toppings"] = customregistry.RESTInPeace(toppingRest)

    apiGroupInfo.VersionedResourcesStorageMap["v1alpha1"] = v1alpha1storage

    v1beta1storage := map[string]rest.Storage{}

    pizzaRest = pizzastorage.NewREST(Scheme, c.GenericConfig.RESTOptionsGetter)
    v1beta1storage["pizzas"] = customregistry.RESTInPeace(pizzaRest)

    apiGroupInfo.VersionedResourcesStorageMap["v1beta1"] = v1beta1storage

    if err := s.GenericAPIServer.InstallAPIGroup(&apiGroupInfo); err != nil {
        return nil, err
    }

    return s, nil
}

```

该 APIGroupInfo 有我们在定制的通用注册表引用“[注册表和战略](#)”通过一个策略。对于每个组版本和资源，我们使用实现的构造函数创建注册表的实例。

该 customregistry.RESTInPeace 包装只是当注册表构造返回一个错误，恐慌帮手：

```

func RESTInPeace(storage rest.StandardStorage, err error) rest.StandardStorage {
    if err != nil {
        err = fmt.Errorf("unable to create REST storage: %v", err)
    }
}

```

```

        panic(err)
    }
    return storage
}

```

注册表本身与版本无关，因为它在内部对象上运行；请参见图8-5。因此，我们为每个版本调用相同的注册表构造函数。

`InstallAPIGroup` 最后的调用将我们引导到一个完整的自定义API服务器，可以为我们的自定义API组提供服务，如图8-7所示。

在完成所有这些繁重的管道之后，是时候看看我们的新API组了。为此，我们启动服务器，如“[第一次启动](#)”中所示。但这次发现信息不是空的，而是显示我们新注册的资源：

```

$ 卷曲-k的https://本地主机:443 / API的
{
  "kind": "APIGroupList",
  "groups": [
    {
      "name": "restaurant.programming-kubernetes.info",
      "versions": [
        {
          "groupVersion": "restaurant.programming-kubernetes.info/v1beta1",
          "version": "v1beta1"
        },
        {
          "groupVersion": "restaurant.programming-kubernetes.info/v1alpha1",
          "version": "v1alpha1"
        }
      ],
      "preferredVersion": {
        "groupVersion": "restaurant.programming-kubernetes.info/v1beta1",
        "version": "v1beta1"
      },
      "serverAddressByClientCIDRs": [
        {
          "clientCIDR": "0.0.0.0/0",
          "serverAddress": ":443"
        }
      ]
    }
  ]
}

```

有了这个，我们几乎达到了服务餐厅API的目标。我们已经连接了API组版本，转换已到位，验证工作正常。

缺少的是检查披萨中提到的顶部确实存在于群集中。我们可以在验证函数中添加它。但传统上这些只是格式验证功能，它们是静态的，不需要运行其他资源。

相反，在接纳中实施更复杂的检查 - 下一节的主题。

入场

一切请求在被解组，默认并转换为内部类型后通过了一系列准入插件；请参阅 [图8-2](#)。更确切地说，请求通过两次入场：

- 变异的插件
- 验证插件

入场插件可以是变异和验证，因此可能会被录取机制调用两次：

- 一旦进入突变阶段，就会依次调用所有变异插件
- 进入验证阶段后，为所有验证插件调用（可能并行化）

更确切地说，插件可以实现变异和验证准入接口，两种情况都有两种不同的方法。

注意

在分离变异和验证之前，每个插件只有一个调用。几乎不可能密切关注每个插件做了哪些突变以及哪些突变因此，允许插入顺序有意义地导致用户的一致行为。

这种两步架构至少可确保在所有插件的最后完成验证，从而保证一致性。

除此之外链（即两个准入阶段的插件顺序）是相同的。始终为两个阶段启用或禁用插件。

入场插件，至少是本章所述的Golang中实现的插件，可以使用内部类型。相比之下， webhook允许插件（参见“[Admission Webhooks](#)”）基于外部类型，并涉及到webhook和back的转换（在变异 webhooks的情况下）。

但毕竟这个理论，让我们进入代码。

履行

准入插件是一种实现：

- 准入插件界面 `Interface`
- 可选的 `MutatingInterface`
- 可选的 `ValidatingInterface`

这三个都可以在包[k8s.io/apiserver/pkg/admission](#)中找到：

```
// Operation is the type of resource operation being checked for
// admission control
type Operation string.

// Operation constants
const (
    Create Operation = "CREATE"
    Update Operation = "UPDATE"
```

```

    Delete Operation = "DELETE"
    Connect Operation = "CONNECT"
}

// Interface is an abstract, pluggable interface for Admission Control
// decisions.
type Interface interface {
    // Handles returns true if this admission controller can handle the given
    // operation where operation can be one of CREATE, UPDATE, DELETE, or
    // CONNECT.
    Handles(operation Operation) bool.
}

type MutationInterface interface {
    Interface

    // Admit makes an admission decision based on the request attributes.
    Admit(a Attributes, o ObjectInterfaces) (err error)
}

// ValidationInterface is an abstract, pluggable interface for Admission Control
// decisions.
type ValidationInterface interface {
    Interface

    // Validate makes an admission decision based on the request attributes.
    // It is NOT allowed to mutate.
    Validate(a Attributes, o ObjectInterfaces) (err error)
}

```

您会看到该 `Interface` 方法 `Handles` 负责对操作进行过滤。变通插件被调用 via `Admit`，验证插件被调用 via `Validate`。

在 `ObjectInterfaces` 可以访问通常由方案实施的助手：

```

type ObjectInterfaces interface {
    // GetObjectCreator is the ObjectCreator for the requested object.
    GetObjectCreator() runtime.ObjectCreator
    // GetObjectTyper is the ObjectTyper for the requested object.
    GetObjectTyper() runtime.ObjectTyper
    // GetObjectDefaulter is the ObjectDefaulter for the requested object.
    GetObjectDefaulter() runtime.ObjectDefaulter
    // GetObjectConvertor is the ObjectConvertor for the requested object.
    GetObjectConvertor() runtime.ObjectConvertor
}

```

传递给插件的属性（通过 `Admit` 或 `Validate` 两者）基本上包含从对实现高级检查很重要的请求中提取的所有信息：

```

// Attributes is an interface used by AdmissionController to get information
// about a request that is used to make an admission decision.
type Attributes interface {
    // GetName returns the name of the object as presented in the request.
    // On a CREATE operation, the client may omit name and rely on the
    // server to generate the name. If that is the case, this method will
    // return the empty string.
    GetName() string
    // GetNamespace is the namespace associated with the request (if any).
    GetNamespace() string
    // GetResource is the name of the resource being requested. This is not the
    // kind. For example: pods.
    GetResource() schema.GroupVersionResource
    // GetSubresource is the name of the subresource being requested. This is a
    // different resource, scoped to the parent resource, but it may have a
    // different kind.
    // For instance, /pods has the resource "pods" and the kind "Pod", while
    // /pods/foo/status has the resource "pods", the sub resource "status", and
    // the kind "Pod" (because status operates on pods). The binding resource for
    // a pod, though, may be /pods/foo/binding, which has resource "pods",
    // subresource "binding", and kind "Binding".
    GetSubresource() string
    // GetOperation is the operation being performed.
    GetOperation() Operation
    // IsDryRun indicates that modifications will definitely not be persisted for
    // this request. This is to prevent admission controllers with side effects
    // and a method of reconciliation from being overwhelmed.
    // However, a value of false for this does not mean that the modification will
    // be persisted, because it could still be rejected by a subsequent
    // validation step.
    IsDryRun() bool
    // GetObject is the object from the incoming request prior to default values
    // being applied.
    GetObject() runtime.Object
    // GetOldObject is the existing object. Only populated for UPDATE requests.
    GetOldObject() runtime.Object
    // GetKind is the type of object being manipulated. For example: Pod.
    GetKind() schema.GroupVersionKind
    // GetUserinfo is information about the requesting user.
    GetUserInfo() user.Info

    // AddAnnotation sets annotation according to key-value pair. The key
    // should be qualified, e.g., podsecuritypolicy.admission.k8s.io/admit-policy,
    // where "podsecuritypolicy" is the name of the plugin, "admission.k8s.io"
    // is the name of the organization, and "admit-policy" is the key
    // name. An error is returned if the format of key is invalid. When
    // trying to overwrite annotation with a new value, an error is
    // returned. Both ValidationInterface and MutationInterface are
    // allowed to add Annotations.
    AddAnnotation(key, value string) error
}

```

```
}
```

在变异的情况下 - 也就是说，在 `Admit(a Attributes) error` 方法的实现中- 属性可以是变异的，或者更确切地说，是从 `GetObject() runtime.Object can` 返回的对象。

在验证案例中，不允许变异。

这两种情况都允许调用 `AddAnnotation(key, value string) error`，这允许我们添加最终在API服务器的审计输出中的注释。这有助于理解入口插件为何突变或拒绝请求。

通过 `nil` 从 `Admit` 或返回非错误来发出拒绝信号 `Validate`。

小费

改变准入插件以验证验证准入阶段的变化是一种很好的做法。原因是其他插件，包括 webhook 准入插件，可能会进一步增加更改。如果准入插件保证满足某些不变量，则只有验证步骤才能确保确实如此。

许可插件必须 `Handles(operation Operation) bool` 从 `admission.Interface` 接口实现该方法。在同一个包中有一个帮助器 `Handler`。它可以通过嵌入到自定义许可插件中来实例化 `NewHandler(ops ...Operation) *Handler` 并实现该 `Handles` 方法 `Handler`：

```
type CustomAdmissionPlugin struct {
    *admission.Handler
    ...
}
```

入场插件应该始终首先检查传递的对象的 `GroupVersionKind`：

```
func (d *PizzaToppingsPlugin) Admit(
    a admission.Attributes,
    o ObjectInterfaces,
) error {
    // we are only interested in pizzas
    if a.GetKind().GroupKind() != restaurant.Kind("Pizza") {
        return nil
    }

    ...
}
```

同样对于验证案例：

```
func (d *PizzaToppingsPlugin) Validate(
    a admission.Attributes,
    o ObjectInterfaces,
) error {
    // we are only interested in pizzas
```

```

if a.GetKind().GroupKind() != restaurant.Kind("Pizza") {
    return nil
}

...
}

```

为什么API服务器管道不会预过滤对象

对于本机许可插件，没有注册机制使得支持对象的信息可用于API服务器机器，以便仅为其支持的对象调用插件。一个原因是Kubernetes API服务器中的许多插件（发明了许可机制）支持大量对象。

完整的示例许可实现如下所示：

```

// Admit ensures that the object in-flight is of kind Pizza.
// In addition checks that the toppings are known.
func (d *PizzaToppingsPlugin) Validate(
    a admission.Attributes,
    _ admission.ObjectInterfaces,
) error {
    // we are only interested in pizzas
    if a.GetKind().GroupKind() != restaurant.Kind("Pizza") {
        return nil
    }

    if !d.WaitForReady() {
        return admission.NewForbidden(a, fmt.Errorf("not yet ready"))
    }

    obj := a.GetObject()
    pizza := obj.(*restaurant.Pizza)
    for _, top := range pizza.Spec.Toppings {
        err := _, err := d.toppingLister.Get(top.Name)
        if err != nil && errors.NotFound(err) {
            return admission.NewForbidden(
                a,
                fmt.Errorf("unknown topping: %s", top.Name),
            )
        }
    }

    return nil
}

```

它采取以下步骤：

1. 检查传递的对象是否正确
2. 在线人准备好之前禁止访问
3. 通过提交者线人验证披萨规范中提到的每个顶部实际上作为 `Topping` 群集中的对象存在

请注意，列表器只是informer内存存储的接口。所以这些 Get 电话会很快。

注册

入场插件必须注册。这是通过一个 `Register` 功能完成的：

```
func Register(plugins *admission.Plugins) {
    plugins.Register(
        "PizzaTopping",
        func(config io.Reader) (admission.Interface, error) {
            return New()
        },
    )
}
```

此功能被添加到插件列表中 `RecommendedOptions`（参见“[选项和配置模式和启动管道](#)”）：

```
func (o *CustomServerOptions) Complete() error {
    // register admission plugins
    pizzatoppings.Register(o.RecommendedOptions.Admission.Plugins)

    // add admisionon plugins to the RecommendedPluginOrder
    oldOrder := o.RecommendedOptions.Admission.RecommendedPluginOrder
    o.RecommendedOptions.Admission.RecommendedPluginOrder =
        append(oldOrder, "PizzaToppings")

    return nil
}
```

在这里，`RecommendedPluginOrder` 列表预填充了通用许可插件，每个API服务器应该保持启用，以成为集群中的良好API约定公民。

最好不要触摸订单。一个原因是获得正确的订单远非微不足道。当然，如果插件行为是严格必要的，那么在列表末尾以外的位置添加自定义插件就可以了。

自定义API服务器的用户将能够以通常的许可禁用自定义许可插件链配置标志（`--disable-admission-plugins` 例如）。默认情况下，我们自己的插件已启用，因为我们没有明确禁用它。

入场可以使用配置文件配置插件。为此，我们解析前面显示 `io.Reader` 的 `Register` 函数的输出。将 `--admission-control-config-file` 允许我们的配置文件传递给插件，像这样：

```
kind: AdmissionConfiguration
apiVersion: apiserver.k8s.io/v1alpha1
plugins:
- name: CustomAdmissionPlugin
  path: custom-admission-plugin.yaml
```

或者，我们可以进行内联配置，将所有入场配置集中在一个地方：

```
kind: AdmissionConfiguration
apiVersion: apiserver.k8s.io/v1alpha1
plugins:
- name: CustomAdmissionPlugin
  configuration:
    your-custom-yaml-inline-config
```

我们简要地提到我们的入场插件使用配料通知器来检查披萨中提到的浇头是否存在。我们还没有谈到如何将它连接到准入插件。我们现在就这样做。

管道资源

入场插件通常需要客户端和线人或其他资源来实现其行为。我们可以使用插件初始化器来完成此资源管道。

有一些标准插件初始化器。如果您的插件想要被他们调用，则必须使用回调方法实现某些接口（有关详细信息，请参阅k8s.io/apiserver/pkg/admission/initializer）：

```
// WantsExternalKubeClientSet defines a function that sets external ClientSet
// for admission plugins that need it.
type WantsExternalKubeClientSet interface {
    SetExternalKubeClientSet(kubernetes.Interface)
    admission.InitializationValidator
}

// WantsExternalKubeInformerFactory defines a function that sets InformerFactory
// for admission plugins that need it.
type WantsExternalKubeInformerFactory interface {
    SetExternalKubeInformerFactory(informers.SharedInformerFactory)
    admission.InitializationValidator
}

// WantsAuthorizer defines a function that sets Authorizer for admission
// plugins that need it.
type WantsAuthorizer interface {
    SetAuthorizer(authorizer.Authorizer)
    admission.InitializationValidator
}

// WantsScheme defines a function that accepts runtime.Scheme for admission
// plugins that need it.
type WantsScheme interface {
    SetScheme(*runtime.Scheme)
    admission.InitializationValidator
}
```

实现其中一些，并在启动期间调用插件，以便访问Kubernetes资源或API服务器全局方案。

此外，`admission.InitializationValidator` 应该实现接口以最终检查插件是否正确设置：

```
// InitializationValidator holds ValidateInitialization functions, which are
// responsible for validation of initialized shared resources and should be
// implemented on admission plugins.
type InitializationValidator interface {
    ValidateInitialization() error
}
```

标准初始化程序很棒，但我们需要访问`toppings informer`。那么，让我们来看看如何添加我们自己的初始化器。初始化程序包括：

- 甲 `Wants*` 接口（例如，`WantsRestaurantInformerFactory`），其应当由承认插件来实现：

```
// WantsRestaurantInformerFactory defines a function that sets
// InformerFactory for admission plugins that need it.
type WantsRestaurantInformerFactory interface {
    SetRestaurantInformerFactory(informers.SharedInformerFactory)
    admission.InitializationValidator
}
```

- 初始化器结构，实现 `admission.PluginInitializer`：

```
func (i restaurantInformerPluginInitializer) Initialize(
    plugin admission.Interface,
) {
    if wants, ok := plugin.(WantsRestaurantInformerFactory); ok {
        wants.SetRestaurantInformerFactory(i.informers)
    }
}
```

换句话说，该 `Initialize()` 方法检查传递的插件是否实现了相应的自定义初始化程序 `Wants*` 接口。如果是这种情况，初始化程序将调用插件上的方法。

- 初始化构造函数的管道（参见“[选项和配置模式和启动管道](#)”）：`RecommendedOptions.Extra\AdmissionInitializers`

```
func (o *CustomServerOptions) Config() (*apiserver.Config, error) {
    ...
    o.RecommendedOptions.ExtraAdmissionInitializers =
        func(c *genericapiserver.RecommendedConfig) (
            []admission.PluginInitializer, error,
        ) {
            client, err := clientset.NewForConfig(c.LoopbackClientConfig)
            if err != nil {
                return nil, err
            }
            return []admission.PluginInitializer{
                &WantRestaurantInformerFactory{informers: c.SharedInformerFactory},
            }, nil
        }
}
```

```

        }
        informerFactory := informers.NewSharedInformerFactory(
            client, c.LoopbackClientConfig.Timeout,
        )
        o.SharedInformerFactory = informerFactory
        return []admission.PluginInitializer{
            custominitializer.New(informerFactory),
        }, nil
    }

    ...
}

```

这段代码创建了一个餐厅API组的loopback客户端，创建一个相应的informer工厂，将其存储在选项中 `o`，并为其返回一个插件初始化程序。

同步线人

如果在接收插件中使用告密者，在实际 `Admit()` 或 `Validate()` 功能中使用它们之前，请务必首先检查告密者是否已同步。`Forbidden` 在此之前拒绝具有错误的请求。

使用“[实现](#)”中 `Handler` 描述的辅助结构，我们可以轻松地使用该函数：`Handler.WaitForReady()`

```

if !d.WaitForReady() {
    return admission.NewForbidden(
        a, fmt.Errorf("not yet ready to handle request"),
    )
}

```

要 `HasSynced()` 在此 `WaitForReady()` 方法中包含自定义的informer方法，请将其从初始化程序实现添加到就绪函数中，如下所示：

```

func (d *PizzaToppingsPlugin) SetRestaurantInformerFactory(
    f informers.SharedInformerFactory) {
    d.toppingLister = f.Restaurant().V1Alpha1().Toppings().Lister()
    d.SetReadyFunc(f.Restaurant().V1Alpha1().Toppings().Informer().HasSynced)
}

```

正如所承诺的那样，准入是完成餐厅API组的自定义API服务器的最后一步。现在我们希望看到它在行动，但不是人为地在本地机器上，而是在真正的Kubernetes集群中。这意味着我们必须查看聚合的自定义API服务器的部署。

部署自定义API服务器

在“[API服务](#)”中，我们看到了该 `APIService` 对象，该对象用于注册自定义API服务器API组版本 Kubernetes API服务器内的聚合器：

```

apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: name
spec:
  group: API-group-name
  version: API-group-version
  service:
    namespace: custom-API-server-service-namespace
    name: custom-API-server-service
    caBundle: base64-caBundle
    insecureSkipTLSVerify: bool
    groupPriorityMinimum: 2000
    versionPriority: 20

```

该 APIService 对象指向服务。通常，此服务将是普通的群集IP服务：即，使用pod将自定义API服务器部署到群集中。该服务将请求转发给pod。

让我们看看Kubernetes清单来实现这一点。

部署清单

我们具有以下清单（[在GitHub上的示例代码中找到](#)），它将成为自定义API服务的集群内部署的一部分：

- 的 APIService 两个版本 v1alpha1 :

```

apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1alpha1.restaurant.programming-kubernetes.info
spec:
  insecureSkipTLSVerify: true
  group: restaurant.programming-kubernetes.info
  groupPriorityMinimum: 1000
  versionPriority: 15
  service:
    name: api
    namespace: pizza-apiserver
    version: v1alpha1

```

.....和 v1beta1 :

```

apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1beta1.restaurant.programming-kubernetes.info
spec:

```

```

insecureSkipTLSVerify: true
group: restaurant.programming-kubernetes.info
groupPriorityMinimum: 1000
versionPriority: 15
service:
  name: api
  namespace: pizza-apiserver
  version: v1alpha1

```

请注意我们设置 `insecureSkipTLSVerify`。这对于开发是可以的，但对于任何生产部署都是不适当。我们将在“[证书和信任](#)”中看到如何解决这个问题。

- 一个 Service 在集群中运行的自定义API服务器实例的面前：

```

apiVersion: v1
kind: Service
metadata:
  name: api
  namespace: pizza-apiserver
spec:
  ports:
    - port: 443
      protocol: TCP
      targetPort: 8443
  selector:
    apiserver: "true"

```

- A Deployment (如此处所示) 或 DaemonSet 自定义API服务器pod：

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: pizza-apiserver
  namespace: pizza-apiserver
  labels:
    apiserver: "true"
spec:
  replicas: 1
  selector:
    matchLabels:
      apiserver: "true"
  template:
    metadata:
      labels:
        apiserver: "true"
    spec:
      serviceAccountName: apiserver
      containers:
        - name: apiserver

```

```

image: quay.io/programming-kubernetes/pizza-apiserver:latest
imagePullPolicy: Always
command: ["/pizza-apiserver"]
args:
- --etcd-servers=http://localhost:2379
- --cert-dir=/tmp/certs
- --secure-port=8443
- --v=4
- name: etcd
  image: quay.io/coreos/etcd:v3.2.24
  workingDir: /tmp

```

- 服务和部署的命名空间：

```

apiVersion: v1
kind: Namespace
metadata:
  name: pizza-apiserver
spec: {}

```

通常，聚合的API服务器被部署到为控制平面pod保留的一些节点，通常称为主节点。在这种情况下，`a DaemonSet` 是每个主节点运行一个自定义API服务器实例的不错选择。这导致高可用性设置。请注意，API服务器是无状态的，这意味着它们可以轻松地多次部署，并且不需要进行领导者选举。

有了这些表现，我们差不多完成了。然而，通常情况下，安全部署需要更多考虑。您可能已经注意到 pod（通过前面的部署定义）使用自定义服务帐户 `apiserver`。这可以通过另一个清单创建：

```

kind: ServiceAccount
apiVersion: v1
metadata:
  name: apiserver
  namespace: pizza-apiserver

```

此服务帐户需要许多权限，我们可以通过RBAC对象添加这些权限。

设置RBAC

该 API 服务的服务帐户首先需要一些通用权限才能参与：

- 命名空间生命周期

只能在现有命名空间中创建对象，并在删除命名空间时删除对象。为此，API 服务器必须获取，列出和查看名称空间。

- 入场 webhooks

经由配置入场网络挂

接 `MutatingWebhookConfigurations` 和 `ValidatingWebhookConfigurations` 独立地从每个API服务器调用。为此，我们的自定义API服务器中的准入机制必须获取，列出和查看这些资源。

我们通过创建RBAC集群角色来配置：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregated-apiserver-clusterrole
rules:
- apiGroups: [""]
  resources: ["namespaces"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["admissionregistration.k8s.io"]
  resources: ["mutatingwebhookconfigurations", "validatingwebhookconfigurations"]
  verbs: ["get", "watch", "list"]
```

并将其绑定到我们的服务帐户 `apiserver` 通过 `ClusterRoleBinding`：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pizza-apiserver-clusterrolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: aggregated-apiserver-clusterrole
subjects:
- kind: ServiceAccount
  name: apiserver
  namespace: pizza-apiserver
```

对于委派身份验证和授权，必须将服务帐户绑定到预先存在的RBAC角色 `extension-apiserver-authentication-reader`：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pizza-apiserver-auth-reader
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: extension-apiserver-authentication-reader
subjects:
- kind: ServiceAccount
  name: apiserver
```

```
namespace: pizza-apiserver
```

和预先存在的RBAC集群角色 `system:auth-delegator` :

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pizza-apiserver:system:auth-delegator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: apiserver
  namespace: pizza-apiserver
```

不安全地运行自定义API服务器

现在 在所有清单和RBAC设置完成后，让我们将API服务器部署到真正的集群。

从签出[GitHub存储库](#)，并配置 `kubectl` 了 `cluster-admin` 特权（这是必需的，因为RBAC规则永远不会升级访问）：

```
$ cd $GOPATH/src/github.com/programming-kubernetes/pizza-apiserver
$ cd artifacts / deployment
$ kubectl apply -f ns.yaml # create the namespace first
$ kubectl apply -f。      # creating all manifests described above
```

现在自定义API服务器正在启动：

```
$ kubectl get pods -A
NAMESPACE NAME READY STATUS AGE
pizza-apiserver pizza-apiserver-7779f8d486-8fpgj 0/2 ContainerCreating 1s
$ # some moments later
$ kubectl get pods -A
pizza-apiserver pizza-apiserver-7779f8d486-8fpgj 2/2 运行75秒
```

什么时候它正在运行，我们仔细检查Kubernetes API服务器是否进行聚合（即代理请求）。首先检查 `APIService` Kubernetes API服务器是否认为我们的自定义API服务器可用：

```
$ kubectl获取apiservices v1alpha1.restaurant.programming-kubernetes.info
名称服务可用
v1alpha1.restaurant.programming-kubernetes.info pizza-apiserver / api True
```

这看起来不错。让我们尝试列出比萨饼，启用日志记录以查看是否出现问题：

```
$ kubectl获得比萨饼--v =7
...
... GET https://localhost:58727/apis?timeout=32s
...
... GET https://localhost:58727/apis/restaurant.programming-kubernetes.info/v1alpha1?超时=32秒
...
... GET https://localhost:58727/apis/restaurant.programming-kubernetes.info/v1beta1/namespaces/default/pizzas?limit=500
... 请求标题:
... 接受: application/json;as=表;v=v1beta1;g=meta.k8s.io, application/json
... 用户代理: kubectl/v1.15.0(darwin/amd64)kubernetes/f873d2a
... 响应状态: 200以6毫秒为单位确定
找不到资源。
```

这看起来非常好。我们看到 `kubectl` 查询发现信息以找出披萨是什么。它查询 `restaurant.programming-kubernetes.info/v1beta1` API 列出比萨饼。不出所料，还没有。但我们当然可以改变：

```
$ cd../examples
$ # install toppings first
$ ls topping * |xargs -n 1kubectl create -f
$ kubectl create -f pizza-margherita.yaml
pizza.restaurant.programming-kubernetes.info/margherita创建
$ kubectl得到披萨-o yaml margherita
apiVersion: restaurant.programming-kubernetes.info/v1beta1
亲切: 披萨
元数据:
  creationTimestamp: "2019-05-05T13:39:52Z"
  名称: 玛格丽塔
  命名空间: 默认
  resourceVersion: "6"
  比萨饼/雏菊
  uid: 42ab6e88-6f3b-11e9-8270-0e37170891d3
规格:
  配料:
    - 名称: 莫扎里拉
      数量: 1
    - 名字: 番茄
      数量: 1
  状态: {}
```

这看起来很棒。但是玛格丽塔披萨很容易。让我们尝试通过创建一个没有列出任何浇头的空比萨来默认行动：

```
apiVersion: restaurant.programming-kubernetes.info/v1alpha1
亲切: 披萨
元数据:
```

名称: 萨拉米香肠

规格:

我们的默认应该把它变成萨拉米香肠披萨萨拉米香肠。我们试试吧:

```
$ kubectl create -f empty-pizza.yaml
pizza.restaurant.programming-kubernetes.info/salami创建
$ kubectl得到披萨-o yaml salami
apiVersion: restaurant.programming-kubernetes.info/v1beta1
亲切: 披萨
元数据:
  creationTimestamp: "2019-05-05T13:42:42Z"
  名称: 萨拉米香肠
  命名空间: 默认
  resourceVersion: "8"
  比萨饼/萨拉米香肠
  uid: a7cb7af2-6f3b-11e9-8270-0e37170891d3
规格:
  配料:
    - 名称: 萨拉米香肠
      数量: 1
    - 名称: 莫扎里拉
      数量: 1
    - 名字: 番茄
      数量: 1
  状态: {}
```

这看起来像一个美味的萨拉米香肠披萨。

现在让我们检查一下我们的自定义插件是否正常工作。我们先删除所有比萨饼和浇头，然后尝试重新制作比萨饼：

```
$ kubectl删除比萨饼 - 全部
pizza.restaurant.programming-kubernetes.info "margherita"已删除
pizza.restaurant.programming-kubernetes.info "salami"删除了
$ kubectl删除顶部--all
topping.restaurant.programming-kubernetes.info "mozzarella"已删除
topping.restaurant.programming-kubernetes.info "salami"已删除
topping.restaurant.programming-kubernetes.info "tomato"已删除
$ kubectl create -f pizza-margherita.yaml
来自服务器的错误(禁止): 创建时出错"pizza-margherita.yaml":
pizzas.restaurant.programming-kubernetes.info "margherita"被禁止:
未知的馅料: 莫扎里拉奶酪
```

没有没有马苏里拉奶酪的玛格丽特，就像任何一家意大利餐厅一样。

看起来我们已经完成了我们在“[示例：比萨餐厅](#)”中描述的内容。但并不完全。安全。再次。我们没有照顾到合适的证书。恶意比萨卖家可能会尝试在我们的用户和自定义API服务器之间进行切换，因为Kubernetes API服务器只接受任何服务证书而不检查它们。我们来解决这个问题。

证书和信托

该 `APIService` 对象包含该 `caBundle` 字段。这配置如何聚合器（在Kubernetes API服务器内）信任自定义API服务器。此CA捆绑包包含用于验证聚合API服务器是否具有其声明的标识的证书（和中间证书）。对于任何严重部署，请将相应的CA捆绑包放入此字段中。

警告

虽然 `insecureSkipTLSVerify` 允许在 `APIService` 禁用认证验证时使用，但在生产设置中使用它是一个坏主意。Kubernetes API服务器将请求发送到受信任的聚合API服务器。设置 `insecureSkipTLSVerify` 为 `true` 表示任何其他actor可以声称是聚合API服务器。这显然是不安全的，不应该在生产环境中使用。

“[委托身份验证和信任](#)”中描述了从自定义API服务器到Kubernetes API服务器的反向信任及其对请求的预身份验证。我们不需要做任何额外的事情。

回到披萨示例：为了确保安全，我们需要服务证书和部署中自定义API服务器的密钥。我们将两者放入一个 `serving-cert` 秘密并将其安装到 `/var/run/apiserver/serving-cert/tls.{crt,key}` 的吊舱中。然后我们使用 `tls.crt` 文件作为CA的 `APIService`。这可以在[GitHub上的示例代码中找到](#)。

证书生成逻辑在[Makefile](#)中编写脚本。

请注意，在实际情况中，我们可能会有某种类型的集群或公司CA，我们可以插入其中 `APIService`。

要查看它的实际效果，请先从新群集开始，或者只是重复使用前一个群集并应用新的安全清单：

```
$ cd ../deployment-secure
$ make
openssl req -new -x509 -subj "/CN=api.pizza-apiserver.svc"
-nodes -newkey rsa: 4096
-keyout tls.key -out tls.crt -days 365
生成4096一点RSA私钥
.....
写新的私钥 'tls.key'
...
$ ls * .yaml |xargs -n 1kubectl apply -f
clusterrolebinding.rbac.authorization.k8s.io/pizza-apiserver:system:auth-delegator不变
rolebinding.rbac.authorization.k8s.io/pizza-apiserver-auth-reader不变
已配置deployment.apps / pizza-apiserver
namespace / pizza-apiserver不变
clusterrolebinding.rbac.authorization.k8s.io/pizza-apiserver-clusterrolebinding不变
clusterrole.rbac.authorization.k8s.io/aggregated-apiserver-clusterrole不变
serviceaccount / apiserver不变
```

```
service / api不变
secret / serving-cert创建
apiservice.apiregistration.k8s.io/v1alpha1.restaurant.programming-kubernetes.info已
配置
apiservice.apiregistration.k8s.io/v1beta1.restaurant.programming-kubernetes.info已配
置
```

请注意 `CN=api.pizza-apiserver.svc` 证书中正确的通用名称。Kubernetes API服务器将请求代理到`api / pizza-apiserver`服务，因此必须将其DNS名称放入证书中。

我们仔细检查我们是否确实禁用了以下 `insecureSkipTLSVerify` 标志 APIService：

```
$ kubectl获取apiservices v1alpha1.restaurant.programming-kubernetes.info -o yaml
apiVersion: apiregistration.k8s.io/v1
kind: APIService
元数据:
  name: v1alpha1.restaurant.programming-kubernetes.info
  ...
  规格:
    caBundle: LS0tLS1C .....
    group: restaurant.programming-kubernetes.info
    groupPriorityMinimum: 1000
    服务:
      名称: api
      命名空间: pizza-apiserver
      版本: v1alpha1
      versionPriority: 15
    状态:
      条件:
        - 最后的过渡时间: "2019-05-05T14:07:07Z"
          消息: 所有检查都已通过
          理由: 通过
          status "True"
          type::可用
        伪影/ deploymen
```

这看起来像预期的那样：`insecureSkipTLSVerify` 已经消失，`caBundle` 字段中填充了我们证书的 `base64` 值并且：该服务仍然可用。

现在让我们看看是否 `kubectl` 仍然可以查询API：

```
$ kubectl获得比萨饼
找不到资源。
$ cd../examples
$ ls topping * |xargs -n 1kubectl create -f
topping.restaurant.programming-kubernetes.info/mozzarella创建
topping.restaurant.programming-kubernetes.info/salami创建
topping.restaurant.programming-kubernetes.info/tomato created
$ kubectl create -f pizza-margherita.yaml
```

`pizza.restaurant.programming-kubernetes.info/margherita`创建

玛格丽塔披萨回来了。这次它完全安全。恶意披萨卖家没有机会开始中间人攻击。Buon appetito!

分享etcd

聚合API使用的服务器 `RecommendOptions`（请参阅“[选项和配置模式和启动管道](#)”）`etcd` 用于存储。这意味着任何自定义API服务器的部署都需要 `etcd` 群集可用。

该集群可以是集群内 - 例如，使用 `etcd` [运营商](#)部署。此运算符允许我们以 `etcd` 声明方式启动和管理集群。运营商将进行更新，上下扩展和备份。这大大减少了操作开销。

或者，`etcd` 群集控制平面的（即，`kube-apiserver`）可以使用。取决于环境 - 自我部署，内部部署或托管服务，如Google容器引擎（GKE） - 这可能是可行的，或者它可能是不可能的，因为用户根本无法访问集群（如果是这样的话）与GKE）。在可行的情况下，自定义API服务器必须使用与Kubernetes API服务器或其他 `etcd` 使用者使用的密钥路径不同的密钥路径。在我们的示例自定义API服务器中，它看起来像这样：

```
const defaultEtcdPathPrefix =
    "/registry/pizza-apiserver.programming-kubernetes.github.com"

func NewCustomServerOptions() *CustomServerOptions {
    o := &CustomServerOptions{
        RecommendedOptions: genericoptions.NewRecommendedOptions(
            defaultEtcdPathPrefix,
            ...
        ),
    }

    return o
}
```

此 `etcd` 路径前缀与使用不同组API名称的Kubernetes API服务器路径不同。

最后但并非最不重要的，`etcd` 可以代理。项目[etcdproxy-controller](#)使用运算符模式实现此机制；也就是说，`etcd` 代理可以自动部署到集群并使用 `EtcdProxy` 对象进行配置。

该 `etcd` 代理会自动完成键映射，所以可以保证 `etcd` 关键前缀不会发生冲突。这使我们可以共享 `etcd` 多个聚合API服务器的集群，而无需担心一个聚合API服务器读取或更改另一个聚合API服务器的数据。这将提高 `etcd` 需要共享群集的环境中的安全性，例如，由于资源限制或避免操作开销。

根据上下文，必须选择其中一个选项。最后，聚合API服务器当然也可以使用其他存储后端，至少在理论上，因为它需要大量自定义代码来实现`k8s.io/apiserver`存储接口。

摘要

这是一个非常大的章节，你把它做到了最后。您已经掌握了很多关于Kubernetes中API的背景知识以及它们的实现方式。

我们看到了自定义API服务器的聚合如何适应Kubernetes集群的体系结构。我们了解了自定义API服务器如何从Kubernetes API服务器接收代理请求。我们已经了解了Kubernetes API服务器如何预先验证这些请求，以及如何使用外部版本和内部版本实现API组。我们学习了如何将对象解码为Golang结构，它们如何被默认，它们如何转换为内部类型，以及它们如何通过许可和验证并最终到达注册表。我们看到了如何将策略插入通用注册表以实现“普通”Kubernetes类REST资源，如何添加自定义许可以及如何使用自定义初始化程序配置自定义许可插件。`APIServices`。我们了解了如何配置RBAC规则以允许自定义API服务器完成其工作。我们讨论了如何`kubectl`查询API组。最后，我们学习了如何使用证书保护与自定义API服务器的连接。

这很多。现在，您可以更好地了解Kubernetes中的API以及它们的实现方式，并希望您有动力去做以下一项或多项：

- 实现您自己的自定义API服务器
- 了解Kubernetes的内部运作方式
- 将来有助于Kubernetes

我们希望您已经找到了一个很好的起点。

[1 正常删除](#)意味着客户端可以通过正常删除期间作为删除调用的一部分。实际的删除是由控制器`kubelet`通过强制删除异步完成（对pod执行的操作）。这样，豆荚有时间干净地关闭。

[2 Kubernetes](#)使用同居来将资源（例如，从`extensions/v1beta1` API组部署）迁移到特定于主题的API组（例如，`apps/v1`）。CRD没有共享存储的概念。

[3](#) 我们将在[第9章中](#)看到，最新Kubernetes版本中提供的CRD转换和录入webhooks也允许我们将这些功能添加到CRD中。

[4](#) PaaS代表平台即服务。

第9章高级自定义资源

在本章中，我们将向您介绍有关CR的高级主题：版本控制，转换和许可控制器。

对于多个版本，CRD变得更加严重，并且与基于Golang的API资源的区别要小得多。当然，同时复杂性在开发和维护方面都有相当大的增长，但在操作上也是如此。我们将这些功能称为“高级”，因为它们将CRD从清单（即纯粹声明性）转移到Golang世界（即进入真正的软件开发项目）。

即使您不打算构建自定义API服务器而是打算直接切换到CRD，我们强烈建议您不要跳过[第8章](#)。围绕高级CRD的许多概念在自定义API服务器的世界中具有直接对应物，并且受它们的推动。阅读[第8章](#)将使本章更容易理解。

此处显示和讨论的所有示例的代码都可以通过[GitHub存储库获得](#)。

自定义资源版本控制

在[第8章中](#)我们了解了如何通过不同的API版本提供资源。在自定义API服务器的示例中，披萨资源存在于版本中 `v1alpha1` 并且 `v1beta1` 同时存在（参见“[示例：比萨餐厅](#)”）。在自定义API服务器内部，请求中的每个对象首先从API端点版本转换为内部版本（请参阅[“内部类型和转换”](#)和[图8-5](#)），然后转换回外部版本进行存储，并转换为回复一个回复。该转换机制由转换函数实现，其中一些是手动编写的，一些是生成的（参见“[转换](#)”）。

版本控制API是一种强大的机制，用于调整和改进API，同时保持旧客户端的兼容性。版本在Kubernetes的每个地方都发挥着核心作用，将alpha API推向beta，最终推广到普通可用性（GA）。在此过程中，API通常会更改结构或进行扩展。

很长一段时间，版本控制只是一个功能[第8章中](#)介绍的聚合API服务器。任何严重的API最终都需要进行版本控制，因为打破与API的使用者的兼容性是不可接受的。

幸运的是，最近在Kubernetes中添加了CRD版本 - 在Kubernetes 1.14中作为alpha版本，并在1.15中升级为beta版。请注意，转换需要结构化的OpenAPI v3验证模式（请参阅[“验证自定义资源”](#)）。结构模式基本上就像Kubebuilder这样的工具。我们将在“[结构模式](#)”中讨论技术细节。

我们将向您展示版本控制如何在这里工作，因为它将在不久的将来在CR的许多严肃应用中发挥核心作用。

修改比萨餐厅

至了解CR转换的工作原理，我们将重新实现[第8章中的](#)披萨餐厅示例，这次仅仅使用CRD - 即没有涉及聚合的API服务器。

对于转换，我们将专注于 `Pizza` 资源：

```
apiVersion: restaurant.programming-kubernetes.info/v1alpha1
kind: Pizza
metadata:
```

```

name: margherita
spec:
  toppings:
    - mozzarella
    - tomato

```

此对象应在 v1beta1 版本中具有不同的浇头切片表示：

```

apiVersion: restaurant.programming-kubernetes.info/v1beta1
kind: Pizza
metadata:
  name: margherita
spec:
  toppings:
    - name: mozzarella
      quantity: 1
    - name: tomato
      quantity: 1

```

在使用时 v1alpha1，重复浇头用于代表额外的奶酪比萨饼，我们 v1beta1 通过使用每个浇头的数量字段来实现这一点。浇头的顺序无关紧要。

我们希望实现这种转换 - 转换 v1alpha1 为 v1beta1 和转换。不过，在我们这样做之前，让我们将 API 定义为 CRD。请注意，我们不能在同一个集群中拥有相同 GroupVersion 的聚合 API 服务器和 CRD。因此，请确保在继续使用 CRD 之前删除 [第8章](#) 中的 APIServices。

```

apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: pizzas.restaurant.programming-kubernetes.info
spec:
  group: restaurant.programming-kubernetes.info
  names:
    kind: Pizza
    listKind: PizzaList
    plural: pizzas
    singular: pizza
  scope: Namespaced
  version: v1alpha1
  versions:
    - name: v1alpha1
      served: true
      storage: true
      schema: ...
    - name: v1beta1
      served: true
      storage: false
      schema: ...

```

CRD定义了两个版本：`v1alpha1` 和 `v1beta1`。我们将前者设置为存储版本（参见图9-1），这意味着要存储的每个对象首先被 etcd 转换为 `v1alpha1`。

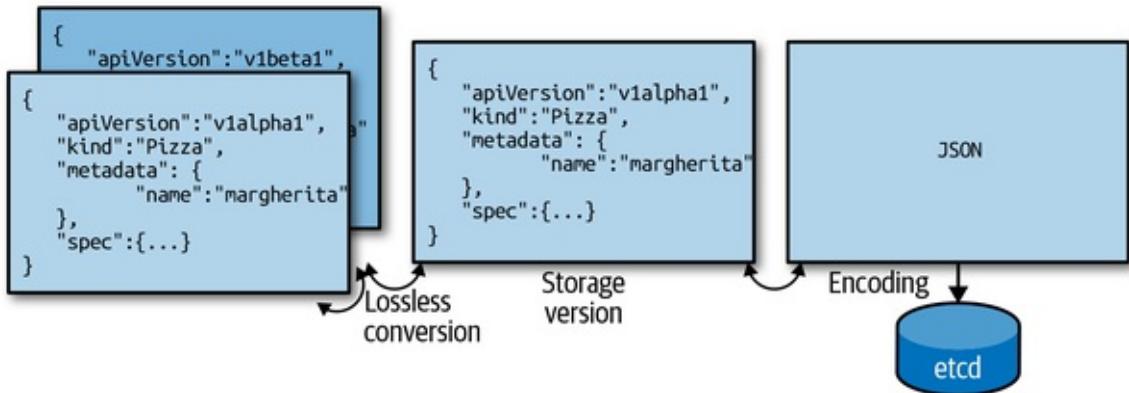


图9-1。转换和存储版本

由于当前定义了CRD，我们可以创建一个对象 `v1alpha1` 并将其检索为 `v1beta1`，但两个API端点都返回相同的对象。这显然不是我们想要的。但我们很快就会改善这一点。

但在我们这样做之前，我们将在群集中设置CRD并创建一个margherita披萨：

```

apiVersion: restaurant.programming-kubernetes.info/v1alpha1
kind: Pizza
metadata:
  name: margherita
spec:
  toppings:
  - mozzarella
  - tomato
  
```

我们注册前面的CRD，然后创建margherita对象：

```

$ kubectl create -f pizza-crd.yaml
$ kubectl create -f margherita-pizza.yaml
  
```

正如所料，我们为两个版本找回了相同的对象：

```

$ kubectl get pizza-margherita -o yaml
apiVersion: restaurant.programming-kubernetes.info/v1beta1
亲切: 披萨
元数据:
  creationTimestamp: "2019-04-14T11:39:20Z"
  一代: 1
  名称: 玛格丽塔
  命名空间: pizza-apiserver
  resourceVersion: "47959"
  selfLink: /apis/restaurant.programming-kubernetes.info/v1beta1/namespaces/pizza-a
  piserver/
  
```

比萨饼/雏菊

uid: f18427f0-5ea9-11e9-8219-124e4d2dc074

规格:

配料:

- 奶酪
- 番茄

Kubernetes使用规范版本顺序; 那是:

- `v1alpha1`

不稳定: 可能会随时离开或更改, 并且通常默认情况下禁用。

- `v1beta1`

走向稳定: 至少在一个版本中平行存在 `v1`; 合同: 没有不兼容的API更改。

- `v1`

稳定或一般可用 (GA) : 将保持良好, 并将兼容。

GA版本按顺序排在第一位, 然后是beta版, 然后是alpha版, 主要版本从高到低排序, 次要版本排序相同。每个不符合此模式的CRD版本都是最后一个, 按字母顺序排序。

在我们的例子中, 前面 `kubectl get pizza` 因此返回 `v1beta1`, 尽管创建的对象是版本 `v1alpha1`。

转换Webhook架构

现在让我们从加入转换 `v1alpha1` 到 `v1beta1` 和背部。CRD转换是通过Kubernetes中的webhook实现的。流程[如图9-2所示](#):

1. 客户端 (例如, 我们的 `kubectl get pizza margherita`) 请求版本。
2. `etcd` 已将对象存储在某个版本中。
3. 如果版本不匹配, 则将存储对象发送到webhook服务器以进行转换。webhook返回转换对象的响应。
4. 转换后的对象将被发送回客户端。

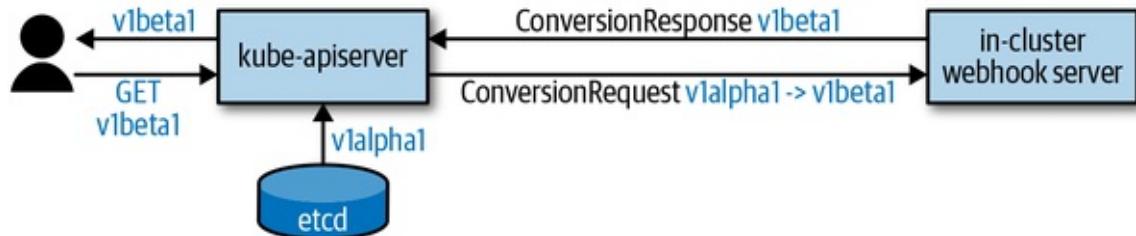


图9-2。转换webhook

我们必须实现这个webhook服务器。在此之前, 让我们看一下webhook API。Kubernetes API服务器发送 `ConversionReview` API组中的对象 `apiextensions.k8s.io/v1beta1`:

```
type ConversionReview struct {
    metav1.TypeMeta `json:",inline"`
    Request *ConversionRequest
    Response *ConversionResponse
}
```

请求字段在发送到webhook的有效负载中设置。响应字段在响应中设置。

请求如下所示：

```
type ConversionRequest struct {
    ...
    // `desiredAPIVersion` is the version to convert given objects to.
    // For example, "myapi.example.com/v1."
    DesiredAPIVersion string

    // `objects` is the list of CR objects to be converted.
    Objects []runtime.RawExtension
}
```

该 `DesiredAPIVersion` 字符串具有 `apiVersion` 我们所知的通常格式 `TypeMeta : group/version`。

`objects` 字段有许多对象。这是一个切片，因为对于一个比萨饼的列表请求， webhook 将收到一个转换请求，该切片是列表请求的所有对象。

webhook 转换并设置响应：

```
type ConversionResponse struct {
    ...
    // `convertedObjects` is the list of converted versions of `request.objects`.
    // if the `result` is successful otherwise empty. The webhook is expected to
    // set apiVersion of these objects to the ConversionRequest.desiredAPIVersion.
    // The list must also have the same size as input list with the same objects
    // in the same order (i.e. equal UIDs and object meta).
    ConvertedObjects []runtime.RawExtension

    // `result` contains the result of conversion with extra details if the
    // conversion failed. `result.status` determines if the conversion failed
    // or succeeded. The `result.status` field is required and represents the
    // success or failure of the conversion. A successful conversion must set
    // `result.status` to `Success`. A failed conversion must set `result.status`
    // to `Failure` and provide more details in `result.message` and return http
    // status 200. The `result.message` will be used to construct an error
    // message for the end user.
    Result metav1.Status
}
```

结果状态告诉Kubernetes API服务器转换是否成功。

但是在请求管道中我们实际调用了转换webhook？我们可以期待什么样的输入对象？为了更好地理解这一点，请查看图9-3中的常规请求管道：所有这些实心和条纹圆都是在k8s.io/apiserver代码中进行转换的地方。

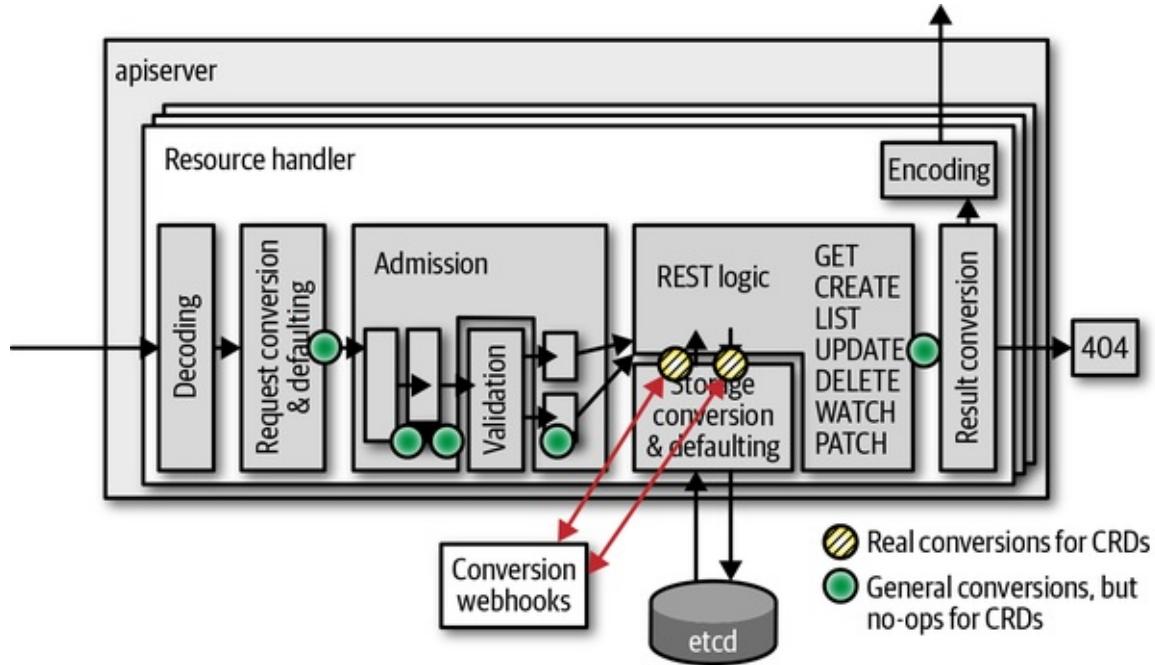


图9-3。转换 webhook 调用CR

与聚合的自定义API服务器（请参阅“[内部类型和转换](#)”）相比，CR不使用内部类型，而是直接在外部API版本之间进行转换。因此，只有那些黄色圆圈实际上在图9-4中进行转换；实心圆圈是CRD的NOOP。换句话说：CRD转换只发生在和从 etcd 。

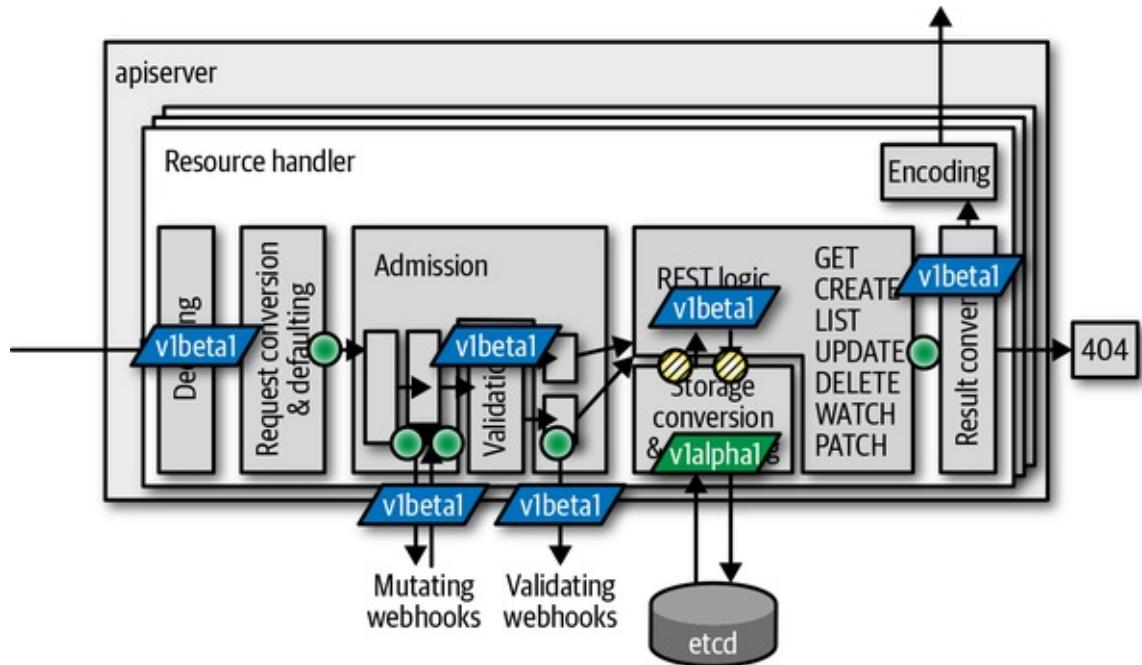


图9-4。CR转换的地方

因此，我们可以假设我们的 webhook 将从请求管道中的这两个位置调用（参见 [图9-3](#)）。

另请注意，修补程序请求会在冲突时自动重试（更新无法重试，并且它们会直接向调用方回复错误）。每次重试都包含读取和写入 `etcd`（[图9-3](#) 中的黄色圆圈），因此每次迭代会导致两次调用 webhook。

警告

[“转化”中](#) 关于转化关键性的所有警告也适用于此：转化必须正确。错误很快导致数据丢失和 API 的不一致行为。

在我们开始实施 webhook 之前，关于 webhook 可以做什么并且必须避免的最后一些建议：

- 请求和响应中对象的顺序不得更改。
- `ObjectMeta` 标签和注释除外不得变异。
- 转换是全部或全部：要么所有对象都成功转换，要么全部失败。

转换Webhook实现

同我们背后的理论，我们准备开始实施 webhook 项目。您可以在 [存储库中](#) 找到源代码，其中包括：

- 作为 HTTPS Web 服务器的 webhook 实现
- 许多端点：
 - `/convert/v1beta1/pizza` 在 `v1alpha1` 和之间转换披萨对象 `v1beta1`。
 - `/admit/v1beta1/pizza` 将该 `spec.toppings` 字段默认为马苏里拉奶酪，番茄，萨拉米香肠。
 - `/validate/v1beta1/pizza` 验证每个指定的顶部是否具有相应的浇头对象。

最后两个端点是入场 webhooks，将在 [“Admission Webhooks”](#) 中详细讨论。相同的 webhook 二进制文件将同时用于准入和转换。

在 `v1beta1` 这些路径不应该与混淆 `v1beta1` 我们的餐厅 API 组，但它意味着作为 `apiextensions.k8s.io` API 组的版本，我们支持作为网络挂接。有一天 `v1` 会支持 webhook API，¹ 此时我们将添加相应的 `v1` 另一个端点，以支持旧的（截至今天）和新的 Kubernetes 集群。可以在 CRD 清单中指定 webhook 支持的版本。

让我们来看看这个转换 webhook 实际上是如何工作的。之后，我们将深入探讨如何将 webhook 部署到真正的集群中。再次注意，webhook 转换在 1.14 中仍然是 alpha，必须使用 `CustomResourceWebhookConversion` 功能门手动启用，但它在 1.15 中以 beta 版的形式提供。

设置HTTPS服务器

该第一步是启动支持传输层安全性或 TLS（即 HTTPS）的 Web 服务器。Kubernetes 中的 Webhooks 需要 HTTPS。转换 webhook 甚至需要 Kubernetes API 服务器针对 CRD 对象中提供的 CA 包成功检查的证书。

在示例项目中，我们使用了作为 `k8s.io/apiserver` 一部分的安全服务库。它提供您可能用于部署 `kube-apiserver` 或聚合 API 服务器二进制文件的所有 TLS 标志和行为。

该 `k8s.io/apiserver` 安全服务代码遵循 `options-config` 模式（请参阅“[选项和配置模式和启动管道](#)”）。将代码嵌入到您自己的二进制文件中非常容易：

```

func NewDefaultOptions() *Options {
    o := &Options{
        *options.NewSecureServingOptions(),
    }
    o.SecureServing.ServerCert.PairName = "pizza-crd-webhook"
    return o
}

type Options struct {
    SecureServing options.SecureServingOptions
}

type Config struct {
    SecureServing *server.SecureServingInfo
}

func (o *Options) AddFlags(fs *pflag.FlagSet) {
    o.SecureServing.AddFlags(fs)
}

func (o *Options) Config() (*Config, error) {
    err := o.SecureServing.MaybeDefaultWithSelfSignedCerts("0.0.0.0", nil, nil)
    if err != nil {
        return nil, err
    }

    c := &Config{}

    if err := o.SecureServing.ApplyTo(&c.SecureServing); err != nil {
        return nil, err
    }

    return c, nil
}

```

在二进制文件的 `main` 函数中，此 `Options` 结构体被实例化并连接到标志集：

```

opt := NewDefaultOptions()
fs := pflag.NewFlagSet("pizza-crd-webhook", pflag.ExitOnError)
globalflag.AddGlobalFlags(fs, "pizza-crd-webhook")
opt.AddFlags(fs)
if err := fs.Parse(os.Args); err != nil {
    panic(err)
}

// create runtime config

```

```

cfg, err := opt.Config()
if err != nil {
    panic(err)
}

stopCh := server.SetupSignalHandler()

...

// run server
restaurantInformers.Start(stopCh)
if doneCh, err := cfg.SecureServing.Serve(
    handlers.LoggingHandler(os.Stdout, mux),
    time.Second * 30, stopCh,
); err != nil {
    panic(err)
} else {
    <-doneCh
}

```

我们用三条路径代替三个点来设置HTTP多路复用器，如下所示：

```

// register handlers
restaurantInformers := restaurantinformers.NewSharedInformerFactory(
    clientset, time.Minute * 5,
)
mux := http.NewServeMux()
mux.Handle("/convert/v1beta1/pizza", http.HandlerFunc(conversion.Serve))
mux.Handle("/admit/v1beta1/pizza", http.HandlerFunc(admission.ServePizzaAdmit))
mux.Handle("/validate/v1beta1/pizza",
    http.HandlerFunc(admission.ServePizzaValidation(restaurantInformers)))
restaurantInformers.Start(stopCh)

```

由于路径上的比萨验证 webhook / validate / v1beta1 / pizza 必须知道集群中现有的顶级对象，我们为 `restaurant.programming-kubernetes.info` API 组实例化一个共享的 informer 工厂。

现在我们来看看后面的实际转换 webhook 实现 `conversion.Serve`。它是一个普通的 GoLang HTTP 处理函数，意味着它获取请求和响应编写器作为参数。

请求正文包含 ConversionReview API 组中的对象 `apiextensions.k8s.io/v1beta1`。因此，我们必须首先从请求中读取正文，然后解码字节切片。我们使用 API Machinery 的解串器来完成此操作：

```

func Serve(w http.ResponseWriter, req *http.Request) {
    // read body
    body, err := ioutil.ReadAll(req.Body)
    if err != nil {
        responsewriters.InternalError(w, req,
            fmt.Errorf("failed to read body: %v", err))
        return
    }
}

```

```

    }

    // decode body as conversion review
    gv := apiextensionsv1beta1.SchemeGroupVersion
    reviewGVK := gv.WithKind("ConversionReview")
    obj, gvk, err := codecs.UniversalDeserializer().Decode(body, &reviewGVK,
        &apiextensionsv1beta1.ConversionReview{})
    if err != nil {
        responsewriters.InternalError(w, req,
            fmt.Errorf("failed to decode body: %v", err))
        return
    }
    review, ok := obj.(*apiextensionsv1beta1.ConversionReview)
    if !ok {
        responsewriters.InternalError(w, req,
            fmt.Errorf("unexpected GroupVersionKind: %s", gvk))
        return
    }
    if review.Request == nil {
        responsewriters.InternalError(w, req,
            fmt.Errorf("unexpected nil request"))
        return
    }

    ...
}

```

此代码使用编解码器工厂 `codecs`，该工厂派生自方案。该方案必须包括 `apiextensions.k8s.io/v1beta1` 的类型。我们还添加了我们的餐厅 API 组的类型。传递的 `ConversionReview` 对象将我们的披萨类型嵌入一个 `runtime.RawExtension` 类型 - 更多关于在一秒钟内。

首先让我们创建我们的方案和编解码器工厂：

```

import (
    apiextensionsv1beta1 "k8s.io/apiextensions-apiserver/pkg/apis/apiextensions/v1b
eta1"
    "github.com/programming-kubernetes/pizza-crd/pkg/apis/restaurant/install"
    ...
)

var (
    scheme = runtime.NewScheme()
    codecs = serializer.NewCodecFactory(scheme)
)

func init() {
    utilruntime.Must(apiextensionsv1beta1.AddToScheme(scheme))
    install.Install(scheme)
}

```

A `runtime.RawExtension` 是嵌入在另一个对象的字段中的类似Kubernetes的对象的包装器。它的结构实际上非常简单：

```
type RawExtension struct {
    // Raw is the underlying serialization of this object.
    Raw []byte `protobuf:"bytes,1,opt,name=raw"`
    // Object can hold a representation of this extension - useful for working
    // with versioned structs.
    Object Object `json:"-"`
}
```

另外，`runtime.RawExtension` 还有特殊的JSON和protobuf编组两种方法。此外，`runtime.Object` 转换为内部类型（即自动编码和解码）时，转换为动态时存在特殊逻辑。

在这种CRD的情况下，我们没有内部类型，因此转换魔法不起作用。仅 `RawExtension.Raw` 填充发送到webhook进行转换的披萨对象的JSON字节切片。因此，我们必须解码这个字节切片。再次注意，一个 `ConversionReview` 可能携带许多对象，这样我们就必须遍历所有对象：

```
// convert objects
review.Response = &apiextensionsv1beta1.ConversionResponse{
    UID: review.Request.UID,
    Result: metav1.Status{
        Status: metav1.StatusSuccess,
    },
}
var objs []runtime.Object
for _, in := range review.Request.Objects {
    if in.Object == nil {
        var err error
        in.Object, _, err = codecs.UniversalDeserializer().Decode(
            in.Raw, nil, nil,
        )
        if err != nil {
            review.Response.Result = metav1.Status{
                Message: err.Error(),
                Status:  metav1.StatusFailure,
            }
            break
        }
    }

    obj, err := convert(in.Object, review.Request.DesiredAPIVersion)
    if err != nil {
        review.Response.Result = metav1.Status{
            Message: err.Error(),
            Status:  metav1.StatusFailure,
        }
    }
}
```

```

        break
    }
    objs = append(objs, obj)
}

```

该 `convert` 调用 `in.Object` 使用所需的API版本作为目标版本进行实际转换。请注意，我们会在第一个错误发生时立即中断循环。

最后，我们 `Response` 在 `ConversionReview` 对象中设置字段，并使用API Machinery的响应编写器将其写回请求的响应主体，该编写器再次使用我们的编解码器工厂来创建序列化器：

```

if review.Response.Result.Status == metav1.StatusSuccess {
    for _, obj := range objs {
        review.Response.ConvertedObjects =
            append(review.Response.ConvertedObjects,
                   runtime.RawExtension{Object: obj},
            )
    }
}

// write negotiated response
responsewriters.WriteObject(
    http.StatusOK, gvk.GroupVersion(), codecs, review, w, req,
)

```

现在，我们必须实施实际的披萨转换。在上面的所有这些管道之后，转换算法是最简单的部分。它只是检查，我们实际上得到了已知版本的比萨饼对象，然后从执行转换 `v1beta1` 到 `v1alpha1` 反之亦然：

```

func convert(in runtime.Object, apiVersion string) (runtime.Object, error) {
    switch in := in.(type) {
    case *v1alpha1.Pizza:
        if apiVersion != v1beta1.SchemeGroupVersion.String() {
            return nil, fmt.Errorf("cannot convert %s to %s",
                                   v1alpha1.SchemeGroupVersion, apiVersion)
        }
        klog.V(2).Infof("Converting %s/%s from %s to %s", in.Namespace, in.Name,
                       v1alpha1.SchemeGroupVersion, apiVersion)

        out := &v1beta1.Pizza{
            TypeMeta: in.TypeMeta,
            ObjectMeta: in.ObjectMeta,
            Status: v1beta1.PizzaStatus{
                Cost: in.Status.Cost,
            },
        }
        out.TypeMeta.APIVersion = apiVersion

        idx := map[string]int{}

```

```

        for _, top := range in.Spec.Toppings {
            if i, duplicate := idx[top]; duplicate {
                out.Spec.Toppings[i].Quantity++
                continue
            }
            idx[top] = len(out.Spec.Toppings)
            out.Spec.Toppings = append(out.Spec.Toppings, v1beta1.PizzaTopping{
                Name: top,
                Quantity: 1,
            })
        }

        return out, nil
    }

    case *v1beta1.Pizza:
        if apiVersion != v1alpha1.SchemeGroupVersion.String() {
            return nil, fmt.Errorf("cannot convert %s to %s",
                v1beta1.SchemeGroupVersion, apiVersion)
        }
        klog.V(2).Infof("Converting %s/%s from %s to %s",
            in.Namespace, in.Name, v1alpha1.SchemeGroupVersion, apiVersion)

        out := &v1alpha1.Pizza{
            TypeMeta: in.TypeMeta,
            ObjectMeta: in.ObjectMeta,
            Status: v1alpha1.PizzaStatus{
                Cost: in.Status.Cost,
            },
        }
        out.TypeMeta.APIVersion = apiVersion

        for i := range in.Spec.Toppings {
            for j := 0; j < in.Spec.Toppings[i].Quantity; j++ {
                out.Spec.Toppings = append(
                    out.Spec.Toppings, in.Spec.Toppings[i].Name)
            }
        }

        return out, nil
    }

    default:
    }
    klog.V(2).Infof("Unknown type %T", in)
    return nil, fmt.Errorf("unknown type %T", in)
}

```

请注意，在转换的两个方向上，我们只需复制 TypeMeta 并将 ObjectMeta API 版本更改为所需的版本，然后转换 topping slice，这实际上是结构上不同的对象的唯一部分。

如果有更多版本，则需要在所有版本之间进行另一次双向转换。或者，当然，我们可以使用中心版本聚合API服务器（请参阅[“内部类型和转换”](#)），而不是实现与所有受支持的外部版本的转换。

部署转换Webhook

我们现在想要部署转换webhook。你可以在[GitHub上](#)找到所有的清单。

CRD的转换webhooks在集群中启动并放在服务对象后面，该服务对象由CRD清单中的转换 webhook 规范引用：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: pizzas.restaurant.programming-kubernetes.info
spec:
  ...
  conversion:
    strategy: Webhook
    webhookClientConfig:
      caBundle: BASE64-CA-BUNDLE
      service:
        namespace: pizza-crd
        name: webhook
        path: /convert/v1beta1/pizza
```

CA捆绑包必须与 webhook 使用的服务证书匹配。在我们的示例项目中，我们使用[Makefile](#)使用 OpenSSL 生成证书，并使用文本替换将它们插入到清单中。

请注意，Kubernetes API服务器假定 webhook 支持所有指定版本的CRD。每个CRD也只有一个这样的 webhook。但由于CRD和转换 webhook 通常由同一个团队拥有，这应该足够了。

另请注意，当前`apiextensions.k8s.io/v1beta1` API 中的服务端口必须为 443。但是，该服务可以将此映射到 webhook pod 使用的任何端口。在我们的示例中，我们将 443 映射到 8443，由 webhook 二进制文件提供服务。

看到行动中的转换

现在 我们了解转换 webhook 如何工作以及它如何连接到集群，让我们看看它的实际运行情况。

我们假设您已经检查了示例项目。此外，我们假设您有一个启用了 webhook 转换的集群（通过 1.14 集群中的功能门或通过 1.15+ 集群，默认情况下启用了 webhook 转换）。获得这样一个集群的一种方法是通过[kind 项目](#)，它提供对 Kubernetes 1.14.1 和本地 `kind-config.yaml` 文件的支持，以启用 webhook 转换的 alpha 功能门（“[Kubernetes 是什么意思？](#)”链接到了一个开发集群的其他选项数量）：

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
kubeadmConfigPatchesJson6902:
```

```

- group: kubeadm.k8s.io
  version: v1beta1
  kind: ClusterConfiguration
  patch: |
    - op: add
      path: /apiServer/extraArgs
      value: {}
    - op: add
      path: /apiServer/extraArgs/feature-gates
      value: CustomResourceWebhookConversion=true

```

然后我们可以创建一个集群:

```

$ kind create cluster --image kindest / node-images: v1.14.1 --config kind-config.yaml
$ export KUBECONFIG=$(kind get kubeconfig-path --name="kind")"

```

现在我们可以部署[我们的清单](#):

```

$ cd pizza-crd
$ cd 清单/部署
$ make
$ kubectl create -f ns.yaml
$ kubectl create -f pizza-crd.yaml
$ kubectl create -f topping-crd.yaml
$ kubectl create -f sa.yaml
$ kubectl create -f rbac.yaml
$ kubectl create -f rbac-bind.yaml
$ kubectl create -f service.yaml
$ kubectl create -f serve-cert-secret.yaml
$ kubectl create -f deployment.yaml

```

这些清单包含以下文件:

- *ns.yaml*

创建 pizza-crd 命名空间。

- 比萨饼*crd.yaml*

指定 `restaurant.programming-kubernetes.info` API组中的披萨资源，包括 `v1alpha1` 和 `v1beta1` 版本以及 webhook 转换配置，如前所示。

- 平顶*crd.yaml*

指定同一API组中的浇头CR，但仅限于 `v1alpha1` 版本。

- *sa.yaml*

介绍 webhook 服务帐户。

- *rbac.yaml*

定义读取，列出和观察浇头的角色。

- *RBAC-bind.yaml*

将早期的RBAC角色绑定到 webhook 服务帐户。

- *service.yaml*

定义 webhook 服务，将webhook pod的端口443映射到8443。

- 服务-*CERT-secret.yaml*

包含webhook pod使用的服务证书和私钥。该证书还可以直接用作前面的披萨CRD清单中的CA捆绑包。

- *deployment.yaml*

启动webhook pods，传递 `--tls-cert-file` 和 `--tls-private-key` 服务证书秘密。

在此之后，我们最终可以创建一个margherita披萨：

```
$ cat ./examples/margherita-pizza.yaml
apiVersion: restaurant.programming-kubernetes.info/v1alpha1
亲切: 披萨
元数据:
  名称: 玛格丽塔
规格:
  配料:
    - 奶酪
    - 番茄
$ kubectl创建./examples/margherita-pizza.yaml
pizza.restaurant.programming-kubernetes.info/margherita创建
```

现在，通过转换webhook，我们可以在两个版本中检索相同的对象。首先明确在 `v1alpha1` 版本中：

```
$ kubectl获取pizzas.v1alpha1.restaurant.programming-kubernetes.info \
  margherita -o yaml
apiVersion: restaurant.programming-kubernetes.info/v1alpha1
亲切: 披萨
元数据:
  creationTimestamp: "2019-04-14T21:41:39Z"
  一代: 1
  名称: 玛格丽塔
  命名空间: pizza-crd
  resourceVersion: "18296"
  比萨饼/雏菊
  uid: 15c1c06a-5efe-11e9-9230-0242f24ba99c
规格:
  配料:
    - 奶酪
```

```
- 番茄
状态: {}
```

然后相同的对象 v1beta1 显示不同的浇头结构:

```
$ kubectl获取pizzas.v1beta1.restaurant.programming-kubernetes.info \
margherita -o yaml
apiVersion: restaurant.programming-kubernetes.info/v1beta1
亲切: 披萨
元数据:
  creationTimestamp: "2019-04-14T21:41:39Z"
  一代: 1
  名称: 玛格丽塔
  命名空间: pizza-crd
  resourceVersion: "18296"
  比萨饼/雏菊
  uid: 15c1c06a-5efe-11e9-9230-0242f24ba99c
规格:
  配料:
    - 名称: 莫扎里拉
      数量: 1
    - 名字: 番茄
      数量: 1
状态: {}
```

同时，在 webhook pod 的日志中，我们看到此转换调用:

```
I0414 21:46:28.639707 1 convert.go:35]转换pizza-crd / margherita
来自restaurant.programming-kubernetes.info/v1alpha1
到restaurant.programming-kubernetes.info/v1beta1
10.32.0.1 - - [14 / Apr / 2019: 21:46:28 +0000]
"POST / convert / v1beta1 / pizza? timeout = 30s HTTP / 2.0"200 968
```

因此， webhook 正在按预期完成其工作。

入场Webhooks

在“[自定义API服务器的用例](#)”中，我们讨论了聚合API服务器比使用CR更好的选择的用例。给出的很多原因是关于使用Golang实现某些行为的自由，而不是限制在CRD清单中的声明性功能。

我们在上一节中已经看到Golang如何用于构建CRD转换webhook。类似的机制用于在Golang中添加CRD的自定义许可。

基本上，我们与聚合API服务器中的自定义许可插件具有相同的自由度（请参阅[“许可”](#)）：存在变异和验证许可的webhook，并且它们在与本机资源相同的位置调用，[如图9所示-5](#)。

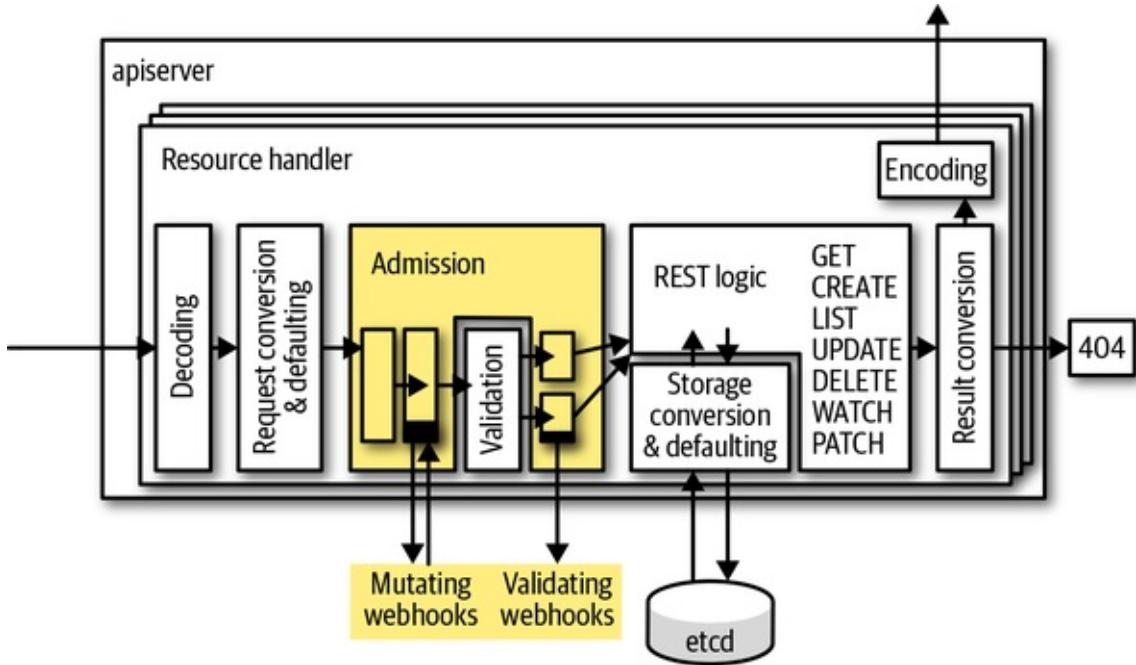


图9-5。CR请求管道中的准入

我们在[“验证自定义资源”](#)中看到了基于OpenAPI的CRD验证。在图9-5中，验证在标记为“验证”的框中完成。之后调用验证准入webhooks，之前是变异录取webhooks。

在配额之前，准入 webhook 几乎在准入插件订单的末尾。入场 webhook 在 Kubernetes 1.14 中是测试版，因此可在大多数集群中使用。

小费

对于入场webhooks API的v1，计划允许最多两次通过入场链。这意味着早期的许可插件或 webhook 可以在一定程度上取决于后来的插件或 webhook 的输出。因此，未来这种机制将变得更加强大。

餐厅示例中的入学要求

该 餐厅示例 使用多项内容：

- `spec.toppings` 如果是 `nil` 莫扎里拉奶酪，西红柿和萨拉米香肠，则默认为空。
- 应该从CR JSON中删除未知字段，而不是持久存储 `etcd`。
- `spec.toppings` 必须仅包含具有相应顶部对象的浇头。

前两个用例是变异的；第三个用例纯粹是验证。因此，我们将使用一个变异 webhook 和一个验证 webhook 来实现这些步骤。

注意

通过[OpenAPI v3验证模式](#)进行本机默认工作正在进行中。OpenAPI有一个 `default` 字段，API服务器将来会应用它。此外，丢弃未知字段将成为每个资源的标准行为，由Kubernetes API服务器通过[称为修剪的机制完成](#)。

修剪在Kubernetes 1.15中以beta版的形式提供。默认计划在1.16中作为测试版提供。当目标集群中的两个功能都可用时，可以在没有任何webhook的情况下实现前面列表中的两个用例。

入场Webhook架构

入场webhooks 在结构上与我们在本章前面看到的转换 webhook非常相似。

它们部署在集群中，在服务映射端口443后面放置到pod的某个端口，并使用 AdmissionReview API组中的审阅对象进行调用 admission.k8s.io/v1beta1：

```
---
// AdmissionReview describes an admission review request/response.
type AdmissionReview struct {
    metav1.TypeMeta `json:",inline"`
    // Request describes the attributes for the admission request.
    // +optional
    Request *AdmissionRequest `json:"request,omitempty"`
    // Response describes the attributes for the admission response.
    // +optional
    Response *AdmissionResponse `json:"response,omitempty"`
}
---
```

它 AdmissionRequest 包含我们用于接纳属性的所有信息（参见“[实现](#)”）：

```
// AdmissionRequest describes the admission.Attributes for the admission request.
type AdmissionRequest struct {
    // UID is an identifier for the individual request/response. It allows us to
    // distinguish instances of requests which are otherwise identical (parallel
    // requests, requests when earlier requests did not modify etc). The UID is
    // meant to track the round trip (request/response) between the KAS and the
    // WebHook, not the user request. It is suitable for correlating log entries
    // between the webhook and apiserver, for either auditing or debugging.
    UID types.UID `json:"uid"`
    // Kind is the type of object being manipulated. For example: Pod
    Kind metav1.GroupVersionKind `json:"kind"`
    // Resource is the name of the resource being requested. This is not the
    // kind. For example: pods
    Resource metav1.GroupVersionResource `json:"resource"`
    // SubResource is the name of the subresource being requested. This is a
    // different resource, scoped to the parent resource, but it may have a
    // different kind. For instance, /pods has the resource "pods" and the kind
    // "Pod", while /pods/foo/status has the resource "pods", the sub resource
    // "status", and the kind "Pod" (because status operates on pods). The
    // binding resource for a pod though may be /pods/foo/binding, which has
    // resource "pods", subresource "binding", and kind "Binding".
    // +optional
    SubResource string `json:"subResource,omitempty"`
}
```

```

// Name is the name of the object as presented in the request. On a CREATE
// operation, the client may omit name and rely on the server to generate
// the name. If that is the case, this method will return the empty string.
// +optional
Name string `json:"name,omitempty"`
// Namespace is the namespace associated with the request (if any).
// +optional
Namespace string `json:"namespace,omitempty"`
// Operation is the operation being performed
Operation Operation `json:"operation"`
// UserInfo is information about the requesting user
UserInfo authenticationv1.UserInfo `json:"userInfo"`
// Object is the object from the incoming request prior to default values
// being applied
// +optional
Object runtime.RawExtension `json:"object,omitempty"`
// OldObject is the existing object. Only populated for UPDATE requests.
// +optional
OldObject runtime.RawExtension `json:"oldObject,omitempty"`
// DryRun indicates that modifications will definitely not be persisted
// for this request.
// Defaults to false.
// +optional
DryRun *bool `json:"dryRun,omitempty"`
}

```

相同的 `AdmissionReview` 对象用于改变和验证准入webhooks。唯一的区别是，在突变的情况下，`AdmissionResponse` 可以有一个字段 `patch` 和 `patchType`，网络挂接已接收到响应之后有要在Kubernetes API服务器内施用。在验证案例中，这两个字段在响应时保持为空。

这里我们目的最重要的领域是 `Object` 字段，它与前面的转换webhook一样 - 使用 `runtime.RawExtension` 类型来存储披萨对象。

我们还获取更新请求的旧对象，并且可以检查是否为只读但在请求中更改的字段。我们在这个例子中没有这样做。但是在Kubernetes中会遇到很多情况，例如，对于pod的大多数组字段，实现了这样的逻辑，因为在创建pod之后你无法更改它的命令。

变异 webhook 返回的补丁必须是 Patch Kubernetes 1.14 中的 JSON 类型（参见 RFC 6902）。此修补程序描述了如何修改对象以满足所需的不变量。

请注意，最佳做法是在最后验证验证 webhook 中的每个变异 webhook 更改，至少如果这些强制属性对于该行为很重要。想象一下，其他一些变异的 webhook 触及对象中的相同字段。然后你不能确定变异的变化会持续到变异入场链的结束。

目前没有订单变异除了字母顺序之外的 webhooks。目前正在讨论，以便在未来以某种方式改变这种状况。

为了验证 webhooks，显然，顺序并不重要，Kubernetes API 服务器甚至会并行调用验证 webhook 以减少延迟。相反，变异 webhooks 会为每个通过它们的请求增加延迟，因为它们是按顺序调用的。

常见的延迟 - 当然严重依赖于环境 - 大约是100毫秒。因此，按顺序运行许多webhook会导致用户在创建或更新对象时会遇到相当大的延迟。

注册入场Webhooks

入场webhooks未在CRD清单中注册。原因是它们不仅适用于CRD，也适用于任何类型的资源。您甚至可以将自定义录取 webhook添加到标准Kubernetes资源中。

而是有注册对象：`MutatingWebhookRegistration` 和 `ValidatingWebhookRegistration`。它们只在种类名称上有所不同：

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: restaurant.programming-kubernetes.info
webhooks:
- name: restaurant.programming-kubernetes.info
  failurePolicy: Fail
  sideEffects: None
  admissionReviewVersions:
  - v1beta1
  rules:
  - apiGroups:
    - "restaurant.programming-kubernetes.info"
    apiVersions:
    - v1alpha1
    - v1beta1
    operations:
    - CREATE
    - UPDATE
    resources:
    - pizzas
  clientConfig:
    service:
      namespace: pizza-crd
      name: webhook
      path: /admit/v1beta1/pizza
  caBundle: CA-BUNDLE
```

这将注册我们 `pizza-crd` 从入院突变对我们资源的两个版本本章开头网络挂接 `pizza` 的API组 `restaurant.programming-kubernetes.info`，和HTTP动词 `CREATE` 及 `UPDATE`（其中包括补丁以及）。

webhook配置中还有其他方法来限制匹配资源 - 例如，命名空间选择器（以排除例如控制平面命名空间以避免引导问题）以及具有通配符和子资源的更高级资源模式。

最后但并非最不重要的是失败模式，可以是 `Fail` 或者 `Ignore`。它指定如果由于其他原因无法访问或失败webhook时要执行的操作。

警告

如果以错误的方式部署，则入场 webhook 可能会破坏群集。允许 webhook 匹配核心类型可以使整个集群无法运行。必须特别注意为非CRD资源调用入场 webhook。

具体来说，最好从 webhook 中排除控制平面和 webhook 资源本身。

实施招生Webhook

同我们在本章开头的转换 webhook 上所做的工作，不难添加录取功能。我们还看到路径 /`admit/v1beta1/pizza` 和 /`validate/v1beta1/pizza` 在 `pizza-crd-webhook` 二进制文件的 `main` 函数中注册：

```
mux.HandleFunc("/admit/v1beta1/pizza", http.HandlerFunc(admission.ServePizzaAdmit))
mux.HandleFunc("/validate/v1beta1/pizza", http.HandlerFunc(
    admission.ServePizzaValidation(restaurantInformers)))
```

两个 HTTP 处理程序实现的第一部分看起来几乎与转换 webhook 相同：

```
func ServePizzaAdmit(w http.ResponseWriter, req *http.Request) {
    // read body
    body, err := ioutil.ReadAll(req.Body)
    if err != nil {
        responsewriters.InternalError(w, req,
            fmt.Errorf("failed to read body: %v", err))
        return
    }

    // decode body as admission review
    reviewGVK := admissionv1beta1.SchemeGroupVersion.WithKind("AdmissionReview")
    decoder := codecs.UniversalDeserializer()
    into := &admissionv1beta1.AdmissionReview{}
    obj, gvk, err := decoder.Decode(body, &reviewGVK, into)
    if err != nil {
        responsewriters.InternalError(w, req,
            fmt.Errorf("failed to decode body: %v", err))
        return
    }
    review, ok := obj.(*admissionv1beta1.AdmissionReview)
    if !ok {
        responsewriters.InternalError(w, req,
            fmt.Errorf("unexpected GroupVersionKind: %s", gvk))
        return
    }
    if review.Request == nil {
        responsewriters.InternalError(w, req,
            fmt.Errorf("unexpected nil request"))
        return
    }
}
```

```
...
}
```

在验证webhook的情况下，我们必须连接informer（用于检查集群中是否存在浇头）。只要未同步informer，我们就会返回内部错误。未同步的线人有不完整的数据，因此可能不知道配料，虽然披萨有效，但披萨会被拒绝：

```
func ServePizzaValidation(informers restaurantinformers.SharedInformerFactory)
    func (http.ResponseWriter, *http.Request)
{
    toppingInformer := informers.Restaurant().V1alpha1().Toppings().Informer()
    toppingLister := informers.Restaurant().V1alpha1().Toppings().Lister()

    return func(w http.ResponseWriter, req *http.Request) {
        if !toppingInformer.HasSynced() {
            responsewriters.InternalError(w, req,
                fmt.Errorf("informers not ready"))
            return
        }

        // read body
        body, err := ioutil.ReadAll(req.Body)
        if err != nil {
            responsewriters.InternalError(w, req,
                fmt.Errorf("failed to read body: %v", err))
            return
        }

        // decode body as admission review
        gv := admissionv1beta1.SchemeGroupVersion
        reviewGVK := gv.WithKind("AdmissionReview")
        obj, gvk, err := codecs.UniversalDeserializer().Decode(body, &reviewGVK,
            &admissionv1beta1.AdmissionReview{})
        if err != nil {
            responsewriters.InternalError(w, req,
                fmt.Errorf("failed to decode body: %v", err))
            return
        }
        review, ok := obj.(*admissionv1beta1.AdmissionReview)
        if !ok {
            responsewriters.InternalError(w, req,
                fmt.Errorf("unexpected GroupVersionKind: %s", gvk))
            return
        }
        if review.Request == nil {
            responsewriters.InternalError(w, req,
                fmt.Errorf("unexpected nil request"))
            return
        }
    }
}
```

```

        ...
    }
}

```

与webhook转换案例一样，我们已经设置了方案和编解码器工厂以及许可API组和我们的餐厅API组：

```

var (
    scheme = runtime.NewScheme()
    codecs = serializer.NewCodecFactory(scheme)
)

func init() {
    utilruntime.Must(admissionv1beta1.AddToScheme(scheme))
    install.Install(scheme)
}

```

有了这两个，我们解码嵌入式披萨对象（这次只有一个，没有切片）`AdmissionReview`：

```

// decode object
if review.Request.Object.Object == nil {
    var err error
    review.Request.Object.Object, _, err =
        codecs.UniversalDeserializer().Decode(review.Request.Object.Raw, nil, nil)
    if err != nil {
        review.Response.Result = &metav1.Status{
            Message: err.Error(),
            Status:  metav1.StatusFailure,
        }
        responsewriters.WriteHeader(http.StatusOK, gvk.GroupVersion(),
            codecs, review, w, req)
        return
    }
}

```

然后我们可以进行实际的变异录入（`spec.toppings` 两个API版本的默认）：

```

orig := review.Request.Object.Raw
var bs []byte
switch pizza := review.Request.Object.Object.(type) {
case *v1alpha1.Pizza:
    // default toppings
    if len(pizza.Spec.Toppings) == 0 {
        pizza.Spec.Toppings = []string{"tomato", "mozzarella", "salami"}
    }
    bs, err = json.Marshal(pizza)
    if err != nil {
        responsewriters.InternalError(w, req,
            fmt.Errorf("unexpected encoding error: %v", err))
    }
}

```

```

        return
    }

    case *v1beta1.Pizza:
        // default toppings
        if len(pizza.Spec.Toppings) == 0 {
            pizza.Spec.Toppings = []v1beta1.PizzaTopping{
                {"tomato", 1},
                {"mozzarella", 1},
                {"salami", 1},
            }
        }
        bs, err = json.Marshal(pizza)
        if err != nil {
            responsewriters.InternalError(w, req,
                fmt.Errorf("unexpected encoding error: %v", err))
            return
        }

    default:
        review.Response.Result = &metav1.Status{
            Message: fmt.Sprintf("unexpected type %T", review.Request.Object.Object),
            Status: metav1.StatusFailure,
        }
        responsewriters.WriteObject(http.StatusOK, gvk.GroupVersion(),
            codecs, review, w, req)
        return
    }
}

```

或者，我们可以使用转换 webhook 中的转换算法，然后仅针对其中一个版本实现默认。这两种方法都是可能的，哪种更有意义取决于上下文。这里，默认很简单，可以实现两次。

最后一步是计算补丁 - 原始对象（`orig` 以 JSON 格式存储）与新默认对象之间的差异：

```

// compare original and defaulted version
ops, err := jsonpatch.CreatePatch(orig, bs)
if err != nil {
    responsewriters.InternalError(w, req,
        fmt.Errorf("unexpected diff error: %v", err))
    return
}
review.Response.Patch, err = json.Marshal(ops)
if err != nil {
    responsewriters.InternalError(w, req,
        fmt.Errorf("unexpected patch encoding error: %v", err))
    return
}
typ := admissionv1beta1.PatchTypeJSONPatch
review.Response.PatchType = &typ
review.Response.Allowed = true

```

我们使用[JSON-Patch库](#)（[Matt Baird的一个带有关键修复的分支](#)）从原始对象 `orig` 和修改后的对象派生补丁 `bs`，两者都作为JSON字节切片传递。或者，我们可以直接操作非类型化的JSON数据并手动创建JSON-Patch。同样，它取决于上下文。使用diff库很方便。

然后，就像在webhook转换中一样，我们通过使用先前创建的编解码器工厂将响应写入响应编写器来结束：

```
responsewriters.WriteObject(
    http.StatusOK, gvk.GroupVersion(), codecs, review, w, req,
)
```

验证 webhook 非常相似，但它使用共享informer中的toppings lister来检查顶部对象是否存在：

```
switch pizza := review.Request.Object.Object.(type) {
case *v1alpha1.Pizza:
    for _, topping := range pizza.Spec.Toppings {
        _, err := toppingLister.Get(topping)
        if err != nil && !errors.NotFound(err) {
            responsewriters.InternalError(w, req,
                fmt.Errorf("failed to lookup topping %q: %v", topping, err))
            return
        } else if errors.NotFound(err) {
            review.Response.Result = &metav1.Status{
                Message: fmt.Sprintf("topping %q not known", topping),
                Status:  metav1.StatusFailure,
            }
            responsewriters.WriteObject(http.StatusOK, gvk.GroupVersion(),
                codecs, review, w, req)
            return
        }
    }
    review.Response.Allowed = true
case *v1beta1.Pizza:
    for _, topping := range pizza.Spec.Toppings {
        _, err := toppingLister.Get(topping.Name)
        if err != nil && !errors.NotFound(err) {
            responsewriters.InternalError(w, req,
                fmt.Errorf("failed to lookup topping %q: %v", topping, err))
            return
        } else if errors.NotFound(err) {
            review.Response.Result = &metav1.Status{
                Message: fmt.Sprintf("topping %q not known", topping),
                Status:  metav1.StatusFailure,
            }
            responsewriters.WriteObject(http.StatusOK, gvk.GroupVersion(),
                codecs, review, w, req)
            return
        }
    }
```

```

    }
    review.Response.Allowed = true
default:
    review.Response.Result = &metav1.Status{
        Message: fmt.Sprintf("unexpected type %T", review.Request.Object.Object),
        Status: metav1.StatusFailure,
    }
}
responsewriters.WriteObject(http.StatusOK, gvk.GroupVersion(),
    codecs, review, w, req)

```

入场Webhook in Action

我们通过在集群中创建两个注册对象来部署两个准入 webhook:

```
$ kubectl create -f validatingadmissionregistration.yaml
$ kubectl create -f mutatingadmissionregistration.yaml
```

在此之后，我们再也无法制作带有未知浇头的比萨饼：

```
$ kubectl create -f ../examples/margherita-pizza.yaml
服务器出错"../examples/margherita-pizza.yaml": 创建时出错:
录取webhook "restaurant.programming-kubernetes.info"否认了请求:
打顶"tomato"未知
```

同时，在 webhook 日志中我们看到：

```
I0414 22:45:46.873541 1 pizzamutation.go:115]违约披萨-dd / in
版本admission.k8s.io/v1beta1, Kind = AdmissionReview
10.32.0.1 - - [14 / Apr / 2019: 22: 45: 46 +0000]
"POST / admit / v1beta1 / pizza? timeout = 30s HTTP / 2.0"200 871
10.32.0.1 - - [14 / Apr / 2019: 22: 45: 46 +0000]
"POST / validate / v1beta1 / pizza? timeout = 30s HTTP / 2.0"200 956
```

在示例文件夹中创建浇头后，我们可以再次创建 margherita 披萨：

```
$ kubectl create -f ../examples/topping-tomato.yaml
$ kubectl create -f ../examples/topping-salami.yaml
$ kubectl create -f ../examples/topping-mozzarella.yaml
$ kubectl create -f ../examples /margherita-pizza.yaml
pizza.restaurant.programming-kubernetes.info/margherita创建
```

最后但并非最不重要的是，让我们检查默认是否按预期工作。我们想要创建一个空的披萨：

```
apiVersion: restaurant.programming-kubernetes.info/v1alpha1
kind: Pizza
```

```
metadata:
  name: salami
spec:
```

这应该是默认为萨拉米香肠披萨，它是：

```
$ kubectl create -f ./examples/empty-pizza.yaml
pizza.restaurant.programming-kubernetes.info/salami created
$ kubectl get pizza salami -o yaml
apiVersion: restaurant.programming-kubernetes.info/v1beta1
kind: Pizza
metadata:
  creationTimestamp: "2019-04-14T22:49:40Z"
  generation: 1
  name: salami
  namespace: pizza-crd
  resourceVersion: "23227"
  uid: 962e2dda-5f07-11e9-9230-0242f24ba99c
spec:
  toppings:
    - name: tomato
      quantity: 1
    - name: mozzarella
      quantity: 1
    - name: salami
      quantity: 1
  status: {}
```

Voilà，一种萨拉米香肠披萨，含有我们所期望的所有配料。请享用！

在结束本章之前，我们希望了解 `apiextensions.k8s.io/v1` CRD 的 API 组版本（即，`nonbeta`，一般可用性） - 即结构模式的引入。

结构模式和CustomResourceDefinitions的未来

从Kubernetes 1.15，OpenAPI v3验证模式（参见“[验证自定义资源](#)”）正在为CRD发挥更重要的作用，因为如果使用任何这些新功能，必须指定模式：

- CRD转换（[见图9-2](#)）
- 修剪（参见“[修剪与保留未知领域](#)”）
- 默认（参见“[默认值](#)”）
- OpenAPI架构[发布](#)

严格地说，模式的定义仍然是可选的，并且每个现有的CRD都将继续工作，但是如果沒有模式，您的CRD将被排除在任何新功能之外。

此外，指定的模式必须遵循某些规则，以强制指定的类型在遵守[Kubernetes API约定](#)的意义上实际上 是理智的。我们称之为结构模式。

结构模式

结构模式是遵循以下规则的OpenAPI v3验证模式（请参阅[“验证自定义资源”](#)）：

1. 模式 `type` 为根，对象节点的每个指定字段（通过 `properties` 或 `additionalProperties` 在 OpenAPI中）以及对于数组节点中的每个项（通过OpenAPI）指定非空类型（`items` 在OpenAPI 中），但以下情况除外：
 - 一个节点 `x-kubernetes-int-or-string: true`
 - 一个节点 `x-kubernetes-preserve-unknown-fields: true`
2. 用于在对象中的每个字段，并在阵列中，这是设置内的每个项目 `allOf`，`anyOf`，`oneOf`，或 `not`，该架构还指定那些逻辑junctions外部的场/项目。
3. 该方案不设 `description`，`type`，`default`，`additionalProperties`，或 `nullable` 内 `allOf`，`anyOf`，`oneOf`，或者 `not`，与两种模式的例外 `x-kubernetes-int-or-string: true`（见[“IntOrString和RawExtensions”](#)）。
4. 如果 `metadata` 指定，则仅限制 `metadata.name` 和 `metadata.generateName` 允许。

这是一个非结构性的例子：

```
properties:
  foo:
    pattern: "abc"
  metadata:
    type: object
    properties:
      name:
        type: string
        pattern: "^a"
      finalizers:
        type: array
        items:
          type: string
          pattern: "my-finalizer"
anyOf:
  - properties:
      bar:
        type: integer
        minimum: 42
      required: ["bar"]
      description: "foo bar object"
```

由于以下违规行为，它不是结构模式：

- 缺少根的类型（规则1）。
- `foo` 缺少的类型（规则1）。
- `bar` 内部 `anyOf` 未指定（规则2）。
- `bar` 的 `type` 是内 `anyOf`（规则3）。
- 描述在 `anyOf`（规则3）内设定。

- `metadata.finalizer` 可能不受限制（规则4）。

相比之下，以下相应的架构是结构性的：

```

type: object
description: "foo bar object"
properties:
  foo:
    type: string
    pattern: "abc"
  bar:
    type: integer
  metadata:
    type: object
    properties:
      name:
        type: string
        pattern: "^a"
anyOf:
- properties:
  bar:
    minimum: 42
required: ["bar"]

```

`NonStructural` 在CRD 的条件下报告违反结构模式规则。

验证自己的的模式 `cnat` 在例如[“确认自定义资源”](#)，并在该模式[比萨饼CRD例子](#)确实是结构。

修剪与保留未知领域

CRD传统上存储任何（可能经过验证的）JSON `etcd`。这意味着未指定字段（如果存在的OpenAPI V3验证架构在所有）将持续。这与本地Kubernetes资源（如pod）形成鲜明对比。如果用户指定了一个字段 `spec.randomField`，那么API服务器HTTPS端点将接受该字段，但在将该pod写入之前将其删除（我们称之为修剪） `etcd`。

如果定义了结构OpenAPI v3验证模式（在全局 `spec.validation.openAPIV3Schema` 或每个版本中），我们可以通过设置 `spec.preserveUnknownFields` 为启用修剪（在创建和更新时删除未指定的字段） `false`。

我们来看看这个 `cnat` 例子。[2使用Kubernetes 1.15集群，我们启用修剪：](#)

```

apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: ats.cnat.programming-kubernetes.info
spec:
  ...
  preserveUnknownFields: false

```

然后我们尝试创建一个具有未知字段的实例：

```
apiVersion: cnat.programming-kubernetes.info/v1alpha1
kind: At
metadata:
  name: example-at
spec:
  schedule: "2019-07-03T02:00:00Z"
  command: echo "Hello, world!"
  someGarbage: 42
```

如果我们检索这个对象 `kubectl get at example-at`，我们看到该 `someGarbage` 值被删除：

```
apiVersion: cnat.programming-kubernetes.info/v1alpha1
kind: At
metadata:
  name: example-at
spec:
  schedule: "2019-07-03T02:00:00Z"
  command: echo "Hello, world!"
```

我们说这 `someGarbage` 已被修剪。

从Kubernetes 1.15开始，`apiextensions/v1beta1`中提供修剪，但默认为关闭；也就是说，`spec.preserveUnknownFields` 默认为 `true`。在`apiextensions/v1`中，`spec.preserveUnknownFields: true` 不允许创建新的CRD。

控制修剪

随着 `spec.preserveUnknownField: false` 在对于该类型和所有版本的所有CR，都启用了CRD修剪。但是，可以通过 `x-kubernetes-preserve-unknown-fields: true` OpenAPI v3验证模式选择不修剪JSON子树：

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
```

该字段 `json` 可以存储任何JSON值，而无需修剪任何内容。

可以部分指定允许的JSON：

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
    type: object
```

```
description: this is arbitrary JSON
```

使用此方法，仅允许对象类型值。

为每个指定的属性（或 `additionalProperties`）再次启用修剪：

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
    type: object
    properties:
      spec:
        type: object
        properties:
          foo:
            type: string
          bar:
            type: string
```

有了这个，价值：

```
json:
  spec:
    foo: abc
    bar: def
    something: x
  status:
    something: x
```

将被修剪为：

```
json:
  spec:
    foo: abc
    bar: def
  status:
    something: x
```

这意味着修剪 `something` 指定 `spec` 对象中的字段（因为指定了“`spec`”），但外部的所有内容都没有。`status` 未指定未修剪的内容。`status.*something*`

IntOrString和RawExtensions

那里结构模式不够表达的情况。其中之一是多态字段 - 可以是不同类型的字段。我们 `IntOrString` 从本地Kubernetes API类型中了解到。

它可以 `IntOrString` 使用 `x-kubernetes-int-or-string: true` 模式中的指令在CRD中使用。同样，`runtime.RawExtensions` 可以使用声明 `x-kubernetes-embedded-object: true`。

例如：

```
type: object
properties:
  intorstr:
    type: object
    x-kubernetes-int-or-string: true
  embedded:
    x-kubernetes-embedded-object: true
    x-kubernetes-preserve-unknown-fields: true
```

这声明：

- 一个名为的字段 `intorstr` 包含整数或字符串
- 一个名为的字段 `embedded` 包含类似Kubernetes的对象，例如完整的pod规范

有关这些指令的所有详细信息，请参阅[CRD官方文档](#)。

我们要讨论的最后一个主题取决于结构模式是默认的。

默认值

在原生Kubernetes类型，通常默认某些值。只有通过改变录取webhooks（参见“[Admission Webhooks](#)”），CRD才能实现默认。但是，从Kubernetes 1.15开始，直接通过上一节中描述的OpenAPI v3架构向CRD 添加了默认支持（参见[设计文档](#)）。

注意

从1.15开始，这仍然是一个alpha功能，这意味着默认情况下它被禁用特色
门 `CustomResourceDefaulting`。但随着升级到beta，可能在1.16，它将在CRD中无处不在。

要默认某些字段，只需通过 `default` OpenAPI v3架构中的关键字指定默认值。在向类型添加新字段时，这非常有用。

cnat 从“[验证自定义资源](#)”中的示例的模式开始，假设我们想要使容器图像可自定义，但默认为 `busybox` 图像。为此，我们将 `image` 字符串类型字段添加到OpenAPI v3架构，并将默认值设置为 `busybox`：

```
type: object
properties:
  apiVersion:
    type: string
  kind:
    type: string
  metadata:
    type: object
```

```

spec:
  type: object
  properties:
    schedule:
      type: string
      pattern: "^\d{4}-([0]\d|1[0-2])-([0-2]\d|3[01])..."
    command:
      type: string
    image:
      type: string
      default: "busybox"
  required:
  - schedule
  - command
status:
  type: object
  properties:
    phase:
      type: string
  required:
  - metadata
  - apiVersion
  - kind
  - spec

```

如果用户在未指定图像的情况下创建实例，则会自动设置该值：

```

apiVersion: cnat.programming-kubernetes.info/v1alpha1
kind: At
metadata:
  name: example-at
spec:
  schedule: "2019-07-03T02:00:00Z"
  command: echo "hello world!"

```

在创建时，它会自动变为：

```

apiVersion: cnat.programming-kubernetes.info/v1alpha1
kind: At
metadata:
  name: example-at
spec:
  schedule: "2019-07-03T02:00:00Z"
  command: echo "hello world!"
  image: busybox

```

这看起来非常方便，并显著改善了CRD的用户体验。而且，`etcd` 从API服务器读取时，所有保留的旧对象将自动继承新字段。³

请注意，`etcd` 不会重写持久化对象（即自动迁移）。换句话说，在读取时，默认值仅在运行时添加，并且仅在由于其他原因更新对象时保留。

摘要

入场和转换webhooks使CRD达到完全不同的水平。在这些功能出现之前，CR主要用于小型，不那么严重的用例，通常用于配置和API兼容性不那么重要的内部应用程序。

使用webhooks，CR看起来更像是本机资源，具有很长的生命周期和强大的语义。我们已经了解了如何实现不同资源之间的依赖关系以及如何设置字段的默认值。

此时，您可能对现有CRD中可以使用这些功能的位置有很多想法。我们很想看到未来基于这些功能的社区创新。

[1](#) `apiextensions.k8s.io` 并且 `admissionregistration.k8s.io` 都计划在Kubernetes 1.16中升级到v1。

[2](#) 我们使用 `cnat` 示例而不是披萨示例，因为前者的结构简单 - 例如，只有一个版本。当然，所有这些都可以扩展到多个版本（即一个模式版本）。

[3](#) 例如，`via kubectl get ats -o yaml`。

附录A.资源

一般

- 官方 [Kubernetes文档](#)
- [GitHub上](#)的Kubernetes社区
- 在 [client-go docs](#) 对Kubernetes松弛实例通道
- [Kubernetes深潜: API服务器 - 第1部分](#)
- [Kubernetes深潜: API服务器 - 第2部分](#)
- [Kubernetes深潜: API服务器 - 第3部分](#)
- [Kubernetes API服务器, 第一部分](#)
- [Kubernetes的力学](#)
- GoDoc for [k8s.io/api](#)

图书

- [Kubernetes: Up and Running, 第2版](#), Kelsey Hightower等。 (O'Reilly) 的
- 与 John Arundel和Justin Domingus (O'Reilly) 合作的[Kubernetes的 Cloud Native DevOps](#)
- Brendan Burns和Craig Tracey管理[Kubernetes](#) (O'Reilly)
- Sébastien Goasguen和Michael Hausenblas (O'Reilly) 的 [Kubernetes Cookbook](#)
- [Kubebuilder书](#)

教程和示例

- [Kubernetes的例子](#)
- [Katacoda Kubernetes游乐场](#)
- [Banzai Cloud Operator SDK](#)
- [操作员开发者指南](#)

用品

- 写一个Kubernetes一直在Golang
- 与Kubernetes告密者保持联系
- 事件, Kubernetes的DNA
- 在Kubernetes事件解释
- Kubernetes中的级别触发和协调
- 比较Kubernetes运算符模式与替代方案
- [Kubernetes运营商](#)
- [Kubernetes自定义资源, 控制器和操作员开发工具](#)
- 使用Operator SDK揭开Kubernetes运营商的神秘面纱: 第1部分

- 在Kubebuilder框架下
- 构建Kubernetes运算符和状态应用程序的最佳实践
- Kubernetes运营商发展指南
- 用slok / kubewebhook改变Webhooks

库

- kubernetes-客户组织
- 州长/州长
- kubernetes / perf-test
- cncf / apisnoop
- open-policy -agent / gatekeeper
- 利益相关者/配置者
- ynqa / kubernetes-rust
- hossainemruz / K8S-初始化, 终结实践
- munnerz / K8S-API寻呼机演示
- m3db / m3db运营商

Index

A

- access control
 - best practices, [Packaging Best Practices](#)
 - for production-ready deployment, [Production-Ready Deployments](#)
 - read access, [Getting the Permissions Right](#)
 - role-based access control (RBAC), [How the API Server Processes Requests](#), [Status subresource](#), [Getting the Permissions Right](#), [Delegated Authorization](#)
 - write access, [Getting the Permissions Right](#)
- admission
 - chain, [Admission](#), [Registering](#)
 - configuration, [Registering](#)
 - initializers, [Options and Config Pattern and Startup Plumbing](#), [Plumbing resources](#)
 - mutating, [Validation](#), [Admission](#)
 - order, [Admission](#)
 - plug-in, [How the API Server Processes Requests](#), [Admission](#), [Registering](#)
 - register, [Registering](#)
 - validating, [Validation](#), [Admission](#)
- admission webhooks
 - architecture, [Admission Webhook Architecture](#)
 - example, [Admission Requirements in the Restaurant Example](#)
 - implementing, [Implementing an Admission Webhook](#)
 - overview of, [Admission Webhooks](#)
 - registering, [Registering Admission Webhooks](#)
 - using, [Admission Webhook in Action](#)
- aggregated API server (see aggregation)
- aggregation, [Custom API Servers](#), [Deploying Custom API Servers](#), [Certificates and Trust](#), [Custom Resource Versioning](#), [Setting Up the HTTPS Server](#)
- aggregator (see aggregation)
- alpha versions, [API Versions and Compatibility Guarantees](#)
- Ansible, [Other Packaging Options](#)
- API groups, [API Terminology](#)
- API Machinery features, [Versioning and Compatibility](#), [API Machinery in Depth-Scheme](#) (see also [Kubernetes API](#))
- API servers (see custom API servers)
- API Services, [API Services](#)
- auditing, [Monitoring, instrumentation, and auditing](#)
- authentication, [Delegated Authentication and Trust](#)
- authorization, [Delegated Authorization](#)
- automated builds, [Production-Ready Deployments](#), [Automated Builds and Testing](#)

B

- Ballerina, [Other Packaging Options](#)
- bearer tokens, [Delegated Authentication and Trust](#)
- beta versions, [API Versions and Compatibility Guarantees](#)
- Borg, [Optimistic Concurrency](#)
- builder pattern, [Creating and Using a Client](#)

C

- caching
 - cache coherency issues, [Informers and Caching](#)
 - in-memory, [Informers and Caching-Informers and Caching](#)
 - work queues, [Work Queue](#)
- categories, [Short Names and Categories](#)
- certificates and trust, [Certificates and Trust](#)
- Chang, Eric, [Other Approaches](#)
- charts, [Helm](#)
- Chef, [Other Packaging Options](#)
- CLI-client based operator creation, [Other Approaches](#)
- client sets
 - client expansion, [Client Expansion](#)
 - client options, [Client Options](#)
 - creating, [Creating and Using a Client](#)
 - discovery client, [Client Sets](#)
 - listings and deletions, [Listings and Deletions](#)
 - main interface, [Client Sets](#)
 - role of, [Client Sets](#)
 - status subresources, [Status Subresources: UpdateStatus](#)
 - versioned clients and internal clients, [Client Sets](#)
 - watches, [Watches](#)
- client-gen tags, [client-gen Tags](#)
- client-go
 - API Machinery repository, [API Machinery in Depth-Scheme](#)
 - client sets, [Client Sets-Client Options](#)
 - custom resource access, [Dynamic Client](#)
 - downloading, [The Client Library](#)
 - informers and caching, [Informers and Caching-Work Queue](#)
 - Kubernetes objects in Go, [Kubernetes Objects in Go-spec and status](#)

- repositories, [The Repositories-API Versions and Compatibility Guarantees](#)
- vendoring, [Vendoring-Go Modules](#)
- versioning scheme, [The Client Library](#)
- clients
 - controller-runtime client, [controller-runtime Client of Operator SDK and Kubebuilder](#)
 - creating and using, [Creating and Using a Client](#)
 - dynamic clients, [Dynamic Client](#)
 - loopback client, [Plumbing resources](#)
 - typed clients, [Typed Clients-Typed client created via client-gen](#)
- cloud-native applications
 - example of, [A Motivational Example](#)
 - types of apps running on Kubernetes, [What Does Programming Kubernetes Mean?](#)
- cloud-native languages, [Other Packaging Options](#)
- cnat (cloud-native at) example, [A Motivational Example](#)
- code examples, obtaining and using, [Using Code Examples](#)
- code generation
 - benefits of, [Automating Code Generation](#)
 - calling code generators, [Calling the Generators](#)
 - client-gen tags, [client-gen Tags](#)
 - controlling with tags, [Controlling the Generators with Tags](#)
 - deepcopy-gen tags, [deepcopy-gen Tags](#)
 - global tags for, [Global Tags](#)
 - informer-gen and lister-gen, [informer-gen and lister-gen](#)
 - local tags for, [Local Tags](#)
 - runtime.Object and DeepCopyObject, [runtime.Object and DeepCopyObject](#)
- cohabitation, [API Terminology](#), [Use Cases for Custom API Servers](#)
- command line interface (CLI), [Using the API from the Command Line](#)-[Using the API from the Command Line](#)
- comments and questions, [How to Contact Us](#)
- commercially available off-the-shelf (COTS) apps, [What Does Programming Kubernetes Mean?](#)
- compatibility
 - compatibility guarantees, [API Versions and Compatibility Guarantees](#)
 - formally guaranteed support matrix, [Versioning and Compatibility](#)
 - versioning and, [Versioning and Compatibility](#)
- configuration management systems, [Other Packaging Options](#)
- conflict errors, [Optimistic Concurrency](#)
- connection errors, [Client Options](#)

- continuous integration (CI), [Automated Builds and Testing](#)
- controller-runtime client, [controller-runtime Client of Operator SDK and Kubebuilder](#)
- controllers and operators
 - changing cluster objects or the external world, [Changing Cluster Objects or the External World](#)
 - control loop, [The Control Loop](#)
 - custom controller scope, [Packaging Best Practices](#)
 - documenting with inline docs, [Packaging Best Practices](#)
 - edge- versus level-driven triggers, [Edge- Versus Level-Driven Triggers](#)
 - events, [Events](#)
 - footprint and scalability of, [Packaging Best Practices](#)
 - functions of, [Controllers and Operators](#)
 - lifecycle management, [Lifecycle Management](#)
 - operators, [Operators](#)
 - optimistic concurrency, [Optimistic Concurrency](#)
 - packaging, [Lifecycle Management and Packaging-Packaging Best Practices](#)
 - production-ready deployments, [Production-Ready Deployments-Monitoring, instrumentation, and auditing](#)
 - writing custom, [Solutions for Writing Operators-Uptake and Future Directions](#)
- conversion

,

Inner Structure of a Custom API Server

,

Internal Types and Conversion

,

API Installation

- conversion-gen, [Conversions](#)
- ConversionReview, [Conversion Webhook Architecture](#)
- function

,

Conversions

- naming pattern, [Conversions](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336849881448>)

- lossless, [API Terminology](#)

- on-the-fly, [API Versions and Compatibility Guarantees](#)
 - webhooks, [Custom Resource Versioning, Conversion Webhook Architecture](#)
 - core group, [TypeMeta](#)
 - CoreOS, [Operators](#)
 - CRUD verbs, [Versioning and Compatibility](#)
 - curl, [Using the API from the Command Line](#)
 - custom API servers
 - architecture
- ,

The Architecture: Aggregation

```
\-
```

Delegated Authorization

- aggregation, [\[The Architecture: Aggregation\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336853032152\)](#)
- API services, [\[API Services\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm463368529986824\)](#)
- delegated authentication and trust, [\[Delegated Authentication and Trust\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336852141112\)](#)
- delegated authorization, [\[Delegated Authorization\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336852082968\)](#)
- inner structure of, [\[Inner Structure of a Custom API Server\]\(https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336852593128\)](#)

- benefits of, [Use Cases for Custom API Servers](#)
 - CRD drawbacks, [Use Cases for Custom API Servers](#)
 - CustomResourceDefinition, [Using Custom Resources-controller-runtime Client of Operator SDK and Kubebuilder](#)
 - deploying
- ,

Deploying Custom API Servers

```
\-
```

Sharing etcd

- certificates and trust, [Certificates and Trust](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336844899352>)
- deployment manifests, [Deployment Manifests](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336845812776>)
- RBAC setup, [Setting Up RBAC](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336845362968>)
- running insecurely, [Running the Custom API Server Insecurely](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336845083592>)
- sharing etcd, [Sharing etcd](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336844740888>)

- example of, [Example: A Pizza Restaurant](#)

- writing

,

Writing Custom API Servers

\-

Plumbing resources

- admission, [Admission](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336847396440>)
- API installation, [API Installation](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336848136040>)
- conversions, [Conversions](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336850093512>)
- defaulting, [Defaulting](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336849639864>)
- existing option structs, [Options and Config Pattern and Startup Plumbing](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336851504824>)
- first start, [The First Start](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336850418104>)
- internal types and conversion, [Internal Types and Conversion](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336850296824>)
- options and config pattern, [Options and Config Pattern and Startup Plumbing](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336851946328>)
- registry and strategy, [Registry and Strategy](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336848631880>)
- roundtrip testing, [Roundtrip Testing](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336849403896>)
- validation, [Validation](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336849106792>)

- writing API types, [Writing the API Types](<https://learning.oreilly.com/library/view/programming-kubernetes/9781492047094/ch08.html#idm46336850241176>)

- custom resource definitions (CRDs)
 - accessing, [A Developer's View on Custom Resources](#)
 - accessing with client-go dynamic client, [Dynamic Client](#)
 - accessing with controller-runtime client, [controller-runtime Client of Operator SDK and Kubebuilder](#)
 - accessing with typed clients, [Typed Clients-Typed client created via client-gen](#)
 - admission webhooks, [Admission Webhooks-Admission Webhook in Action](#)
 - availability of, [Using Custom Resources](#)
 - best practices, [Packaging Best Practices](#)
 - defining, [Using Custom Resources](#)
 - discovery information, [Discovery Information](#)
 - limits of, [Use Cases for Custom API Servers](#)
 - printer columns, [Printer Columns](#)
 - role of, [Operators, Using Custom Resources](#)
 - short names and categories, [Short Names and Categories](#)
 - structural schemas, [Structural Schemas and the Future of CustomResourceDefinitions-Default Values](#)
 - subresources, [Subresources-Scale subresource](#)
 - type definitions, [Type Definitions](#)
 - validating custom resources, [Validating Custom Resources](#)
 - versioning, [Custom Resource Versioning-Seeing Conversion in Action](#)
 - writing with code generators, [Automating Code Generation-Summary](#)
- custom resources (CR) (see custom resource definitions (CRDs))

D

- declarative state management, [Declarative State Management](#)
- decoding, [Inner Structure of a Custom API Server](#)
- deep copies, [deepcopy-gen Tags-runtime.Object and DeepCopyObject](#)
- deep copy, [Kubernetes Objects in Go](#)
- deep-copy
 - deep-copy methods, [Golang package structure](#)
 - deepcopy-gen, [Calling the Generators](#)
- DeepCopyObject tag, [runtime.Object and DeepCopyObject](#)
- defaulting

,

Internal Types and Conversion

,

Defaulting

,

API Installation

,

Default Values

- defaulter-gen, [Calling the Generators](#)
- delegated authentication, [Delegated Authentication and Trust](#)
- delegated authorization, [Delegated Authorization](#)
- dep (vendor tool), [dep](#)
- deployment (controllers and operators)
 - access control, [Getting the Permissions Right](#)
 - automated builds and testing, [Automated Builds and Testing](#)
 - custom controller observability, [Custom Controllers and Observability](#)
 - lifecycle management, [Lifecycle Management](#)
 - overview of, [Lifecycle Management and Packaging](#)
 - packaging best practices, [Packaging Best Practices](#)
 - packaging challenges, [Packaging: The Challenge](#)
 - packaging with Helm, [Helm](#)
 - packaging with Kustomize, [Kustomize](#)
 - packaging with other tools, [Other Packaging Options](#)
 - production-ready overview, [Production-Ready Deployments](#)
- deployment (custom API servers)
 - certificates and trust, [Certificates and Trust](#)
 - deployment manifests, [Deployment Manifests](#)
 - RBAC setup, [Setting Up RBAC](#)
 - running insecurely, [Running the Custom API Server Insecurely](#)
 - sharing etcd, [Sharing etcd](#)
- desired state, [Declarative State Management](#)
- discovery

,

Short Names and Categories

,

controller-runtime Client of Operator SDK and Kubebuilder

- endpoint, [The First Start](#)
- RESTMapper, [REST Mapping](#)
- discovery client, [Client Sets](#)
- discovery mechanism, [Discovery Information](#)
- distributed version control, [Technology You Need to Understand](#)
- dynamic clients, [Dynamic Client](#)

E

- edge-driven triggers, [Edge- Versus Level-Driven Triggers](#)
- encoding, [Inner Structure of a Custom API Server](#)
- errors
 - advanced error behavior of informers, [Informers and Caching](#)
 - cache coherency issues, [Informers and Caching](#)
 - conflict errors, [Optimistic Concurrency](#)
 - connection errors, [Client Options](#)
 - coping with trigger errors, [Edge- Versus Level-Driven Triggers](#)
- event handlers, [Informers and Caching](#)
- event producers, [Edge- Versus Level-Driven Triggers](#)
- event sources, [Edge- Versus Level-Driven Triggers](#)
- events
 - overview of, [Events](#)
 - watch events versus event objects, [Events](#)
- extension patterns
 - aggregated API servers, [Custom API Servers-Sharing etcd](#)
 - custom resource definitions (CRDs), [Using Custom Resources-controller-runtime Client of Operator SDK and Kubebuilder](#)
 - overview of, [Extension Patterns](#)
- external version, [Internal Types and Conversion](#)

F

- feature gate, [Options and Config Pattern and Startup Plumbing, Default Values](#)
- field selector, [Listings and Deletions](#)
- Flant's Shell operator, [Other Approaches](#)
- fuzzers, [Roundtrip Testing](#)

G

- general availability (GA), [Custom Resource Versioning](#)
- generator
 - client-gen, [Status Subresources: UpdateStatus, Scheme, Global Tags](#)

- conversion-gen, [Conversions](#)
- deepcopy-gen, [Calling the Generators](#)
- defaulter-gen, [Calling the Generators](#)
- informer-gen, [informer-gen and lister-gen](#)
- lister-gen, [informer-gen and lister-gen](#)
- generic registry, [Generic registry](#)
- Git, [Technology You Need to Understand](#)
- glide (vendor tool), [glide](#)
- global tags, [Global Tags](#)
- Go modules, [Go Modules](#)
- Go programming language, [Technology You Need to Understand](#), [What Does Programming Kubernetes Mean?](#), [Basics of client-go](#) (see also [client-go](#))
- Go types, [Kubernetes API Types](#), [TypeMeta](#)
- Golang package structure, [Golang package structure](#)
- Golang types, [API Terminology](#), [Typed Clients](#)
- graceful shutdowns, [Client Options](#)
- graceful termination, [Use Cases for Custom API Servers](#)
- GroupVersion (GV), [Client Sets](#)
- GroupVersionKind (GVK), [API Terminology](#), [Kubernetes Objects in Go](#), [Kinds](#), [Conversions](#), [Implementation](#)
- GroupVersionResource (GVR), [API Terminology](#), [Resources](#)

H

- handler chain, [Inner Structure of a Custom API Server](#)
- health checks, [Production-Ready Deployments](#)
- Helm, [Helm](#)
- HTTP interface, [The HTTP Interface of the API Server](#), [Kubernetes API Versioning](#), [How the API Server Processes Requests](#)-[How the API Server Processes Requests](#), [The Client Library](#)
- hub version, [Internal Types and Conversion](#)

I

- impersonation, [Inner Structure of a Custom API Server](#)
- in-cluster config, [Creating and Using a Client](#)
- informer-gen, [informer-gen and lister-gen](#)
- informers
 - caching and, [Informers and Caching](#)-[Informers and Caching](#)
 - overview of, [The Control Loop](#)
 - syncing, [Plumbing resources](#)
 - work queues, [Work Queue](#)
- internal version, [Internal Types and Conversion](#)
- IntOrString, [IntOrString](#) and [RawExtensions](#)

K

- kinds
 - categories of, [API Terminology](#)
 - formatting of, [Kinds](#)
 - function of, [API Terminology](#)
 - GroupVersionKind (GVK), [API Terminology](#), [Kinds](#)
 - kinds versus resources, [API Terminology](#)
 - living in multiple API groups, [API Terminology](#)
 - relation to Go type, packages, and group names, [TypeMeta](#)
 - typed clients and, [Anatomy of a type](#)
- klog, [Logging](#)
- ksonnet, [Other Packaging Options](#)
- kube-aggregator, [The Architecture: Aggregation](#), [Delegated Authorization](#)
- kube-apiserver, [Type Definitions](#), [The Architecture: Aggregation](#), [Delegated Authorization](#), [Sharing etcd](#), [Setting Up the HTTPS Server](#)
- kube-controller-manager, [Events](#), [Informers and Caching](#)
- kube-dns, [Using the API from the Command Line](#)
- kube-scheduler, [Events](#)
- kube-system, [Events](#), [Using the API from the Command Line](#)
- Kubebuilder
 - additional resources, [Kubebuilder](#)
 - base directory, [Bootstrapping](#)
 - bootstrapping, [Bootstrapping](#)
 - business logic, [Business Logic-Business Logic](#)
 - commands
 - kubebuilder create api, [Bootstrapping](#)
 - kubebuilder init, [Bootstrapping](#)
 - controller-runtime client of, [controller-runtime Client of Operator SDK and Kubebuilder](#)
 - create api command, [Bootstrapping](#)
 - custom resource definition, [Bootstrapping](#)
 - custom resource installation and validation, [Bootstrapping](#)
 - dedicated namespace creation, [Bootstrapping](#)
 - local operator launch, [Bootstrapping](#)
 - versions, [Kubebuilder](#)
- kubeconfig, [Creating and Using a Client](#)
- kubectl, [Events](#), [The API Server](#), [Using the API from the Command Line](#), [Creating and Using a Client](#), [Discovery Information](#), [Validating Custom Resources](#)
- kubectl api-resources, [Using the API from the Command Line](#), [Short Names and Categories](#)
- kubectl api-versions, [Using the API from the Command Line](#)
- kubectl apply, [Packaging: The Challenge](#), [Kustomize](#)
- kubectl apply -f, [Bootstrapping](#), [Bootstrapping](#), [Bootstrapping](#)
- kubectl create -f, [Running the Custom API Server Insecurely](#)
- kubectl delete, [Running the Custom API Server Insecurely](#)

- [kubectl get --raw](#), [Using the API from the Command Line](#)
- [kubectl get apiservices](#), [Running the Custom API Server Insecurely](#)
- [kubectl get at example-at](#), [Pruning Versus Preserving Unknown Fields](#)
- [kubectl get at,pods](#), [Business Logic](#)
- [kubectl get crds](#), [Bootstrapping](#), [Bootstrapping](#), [Bootstrapping](#)
- [kubectl get ds](#), [Short Names and Categories](#)
- [kubectl get pod](#), [Defaulting](#)
- [kubectl logs](#), [Logging](#)
- [kubectl logs example-at-pod](#), [Business Logic](#)
- [kubectl proxy](#), [Using the API from the Command Line](#)
- [kubectl scale --replicas 3](#), [Scale subresource](#)
- [kubecuddler](#), [Other Approaches](#)
- [kubelet](#), [Events](#), [Optimistic Concurrency](#), [Declarative State Management](#)
- [Kubernetes](#)
 - [additional resources](#), [What Does Programming Kubernetes Mean?](#), [The Control Loop](#), [Events](#), [Edge- Versus Level-Driven Triggers](#), [General](#)
 - [API versioning](#), [Kubernetes API Versioning](#)
 - [controllers and operators](#), [Controllers and Operators-Operators](#)
 - [documentation](#), [A Motivational Example](#), [Versioning and Compatibility](#)
 - [ecosystem for](#), [Ecosystem](#)
 - [extension patterns](#), [Extension Patterns](#)
 - [local development environment](#), [What Does Programming Kubernetes Mean?](#)
 - [meaning of programming Kubernetes](#), [What Does Programming Kubernetes Mean?](#)
 - [native app example](#), [A Motivational Example](#)
 - [optimistic concurrency in](#), [Optimistic Concurrency](#)
 - [prerequisites to learning](#), [Technology You Need to Understand](#)
 - [programming in Go](#), [What Does Programming Kubernetes Mean?](#)
 - [types of apps running on](#), [What Does Programming Kubernetes Mean?](#)
 - [versions discussed](#), [Ecosystem](#)
- [Kubernetes API](#)
 - [API Machinery repository](#), [API Machinery](#)
 - [API versioning](#), [Kubernetes API Versioning](#), [Versioning and Compatibility](#)
 - [architecture and core responsibilities](#), [The API Server](#)
 - [benefits of](#), [What Does Programming Kubernetes Mean?](#)
 - [command line control](#), [Using the API from the Command Line](#)-[Using the API from the Command Line](#)
 - [declarative state management](#), [Declarative State Management](#)
 - [Go types repository](#), [Kubernetes API Types](#)
 - [HTTP interface of](#), [The HTTP Interface of the API Server](#)
 - [request processing](#), [How the API Server Processes Requests](#)-[How the API Server Processes Requests](#), [Client Options](#)
 - [terminology](#), [API Terminology](#)-[API Terminology](#)
- [Kubernetes objects in Go](#)
 - [ObjectMeta](#), [ObjectMeta](#)

- overview of, [Kubernetes Objects in Go](#)
- spec and a status section, [spec and status](#)
- TypeMeta, [TypeMeta](#)
- KUDO, [Other Approaches](#)
- Kustomize, [Kustomize](#)
- kutil, [Other Approaches](#)

L

- label selector, [Listings and Deletions](#)
- leader-follower/standby model, [Production-Ready Deployments](#)
- least-privileges principle, [Getting the Permissions Right](#)
- legacy group, [TypeMeta](#)
- level-driven triggers, [Edge- Versus Level-Driven Triggers](#)
- lifecycle management, [Lifecycle Management](#)
- lister-gen, [informer-gen](#) and [lister-gen](#)
- local development environment, [What Does Programming Kubernetes Mean?](#)
- local tags, [Local Tags](#)
- logging, [Production-Ready Deployments](#), [Logging](#)
- long-running requests, [Client Options](#)

M

- manifest files, [Kustomize](#)
- masters, [Deployment Manifests](#)
- Metacontroller, [Other Approaches](#)
- metadata, [ObjectMeta](#)
- monitoring and logging, [Production-Ready Deployments](#), [Custom Controllers and Observability](#)
- mutating plug-ins, [Admission](#)

O

- ObjectTyper, [Strategy](#)
- ObjectMeta, [ObjectMeta](#)
- OLM (Operator Lifecycle Management), [Lifecycle Management](#)
- Omega (Google research paper), [Optimistic Concurrency](#)
- OpenAPI schema language, [Validating Custom Resources](#)
- Operator SDK
 - additional resources, [Business Logic](#)
 - bootstrapping, [Bootstrapping](#)
 - business logic, [Business Logic](#)

- controller-runtime client of, [controller-runtime Client of Operator SDK and Kubebuilder](#)
- installing, [The Operator SDK](#)
- OperatorHub.io, [Operators](#)
- operators

(

see

also controllers and operators)

- building with Operator SDK, [The Operator SDK-Business Logic](#)
- following sample-controller, [Following sample-controller-Business Logic](#)
- implementing with Kubebuilder, [Kubebuilder-Business Logic](#)
- other approaches to writing, [Other Approaches](#)
- overview of, [Operators](#)
- preparation for writing, [Preparation](#)
- writing custom, [Solutions for Writing Operators](#)
- optimistic concurrency, [Optimistic Concurrency](#)
- option-config pattern, [Options and Config Pattern and Startup Plumbing](#), [Setting Up the HTTPS Server](#)

P

- package management, [Technology You Need to Understand](#), [Kubernetes API Types](#), [dep](#), [Helm](#)
- packaging
 - best practices, [Packaging Best Practices](#)
 - challenges of, [Packaging: The Challenge](#)
 - lifecycle management, [Lifecycle Management](#)
 - other options for, [Other Packaging Options](#)
 - with Helm, [Helm](#)
 - with Kustomize, [Kustomize](#)
- parallel scheduler architecture, [Optimistic Concurrency](#)
- Plumi, [Other Packaging Options](#)
- polling, [Edge- Versus Level-Driven Triggers](#)
- post-start hook, [Options and Config Pattern and Startup Plumbing](#)
- printer columns, [Printer Columns](#)
- priority queues, [Work Queue](#)
- Prometheus, [Monitoring, instrumentation, and auditing](#)
- protocol buffers (protobuf), [Creating and Using a Client](#)
- pruning, [Pruning Versus Preserving Unknown Fields](#)
- Puppet, [Other Packaging Options](#)

Q

- questions and comments, [How to Contact Us](#)

R

- rate limiting, [Client Options](#)
- read access, [Getting the Permissions Right](#)
- reflection, [Scheme](#)
- registry, [Options and Config Pattern and Startup Plumbing](#)
- relist period, [Informers and Caching](#)
- remote procedure calls (RPCs), [Events](#)
- replica integer value, [Scale subresource](#)
- repositories
 - API Machinery, [API Machinery](#)
 - API versions and compatibility guarantees, [API Versions and Compatibility Guarantees](#)
 - client library, [The Client Library](#)
 - creating and using clients, [Creating and Using a Client](#)
 - importing, [The Repositories](#)
 - Kubernetes API Go types, [Kubernetes API Types](#)
 - third-party applications, [Vendoring](#)
 - versioning and compatibility, [Versioning and Compatibility](#)
- request processing, [How the API Server Processes Requests](#)-[How the API Server Processes Requests](#), [Client Options](#)
- resource version, [Optimistic Concurrency](#)
- resource version conflict errors, [Optimistic Concurrency](#)
- resources
 - example Kubernetes API space, [API Terminology](#)
 - formatting of, [Resources](#)
 - GroupVersionResource (GVR), [API Terminology](#), [Resources](#)
 - namespaces versus cluster-scoped, [Resources](#)
 - overview of, [API Terminology](#)
 - resources versus kinds, [API Terminology](#)
 - subresources, [API Terminology](#)
- REST client, [Client Sets](#)
- REST config, [Client Sets](#), [Informers and Caching](#), [Dynamic Client](#)
- REST mapping, [API Terminology](#), [REST Mapping](#)
- REST verbs, [The Client Library](#)
- resync period, [Informers and Caching](#)
- role-based access control (RBAC), [How the API Server Processes Requests](#), [Status subresource](#), [Getting the Permissions Right](#), [Delegated Authorization](#), [Setting Up RBAC](#)
- Rook operator kit, [Other Approaches](#)
- roundtrippable conversion, [Internal Types and Conversion](#), [Roundtrip Testing](#)
- runtime.Object, [Kubernetes Objects in Go](#), [Scheme](#), [runtime.Object and DeepCopyObject](#)

S

- Salt, [Other Packaging Options](#)
- sample-controller
 - bootstrapping, [Bootstrapping](#)
 - business logic implementation, [Business Logic-Business Logic](#)
 - implementing operators following, [Following sample-controller](#)
- scale subresource, [Scale subresource](#)
- schema, structural, [Structural Schemas and the Future of CustomResourceDefinitions](#)
- schemes, [Scheme](#)
- semantic versioning (semver), [Versioning and Compatibility](#), [Go Modules](#)
- server request processing, [How the API Server Processes Requests](#)-[How the API Server Processes Requests](#), [Client Options](#)
- server-side printing, [Printer Columns](#)
- service account, [Deployment Manifests](#)
- shared informer factory, [Informers and Caching](#)
- short names, [Short Names and Categories](#)
- Site Reliability Engineers (SREs), [Operators](#)
- spec and a status section, [spec and status](#), [Status subresource](#)
- specifications (specs), [Declarative State Management](#)
- state change
 - declarative state management, [Declarative State Management](#)
 - detecting, [Edge- Versus Level-Driven Triggers](#)
- status (observed state), [Declarative State Management](#)
- status subresources, [Status Subresources: UpdateStatus](#), [Status subresource](#)
- storage versions, [API Versions and Compatibility Guarantees](#)
- stores, [Informers and Caching](#)
- strategy, [Strategy](#)
- structural schemas
 - controlling pruning, [Controlling Pruning](#)
 - default values, [Default Values](#)
 - IntOrString and RawExtensions, [IntOrString and RawExtensions](#)
 - overview of, [Structural Schemas and the Future of CustomResourceDefinitions](#)
 - pruning versus preserving unknown fields, [Pruning Versus Preserving Unknown Fields](#)
- subject access review, [Delegated Authorization](#), [Generic registry](#)
- subresources, [API Terminology](#), [Subresources](#)-[Scale subresource](#)

T

- testing, [Automated Builds and Testing](#)
- third-party applications, [Vendoring](#)
- throttling

,

Client Options

- burst, [Client Options](#)
- queries per second, [Client Options](#)
- timeouts, [Client Options](#)
- triggers
 - coping with errors, [Edge- Versus Level-Driven Triggers](#)
 - edge- versus level-driven triggers, [Edge- Versus Level-Driven Triggers](#)
- type definitions, [Type Definitions](#)
- type system, [API Machinery in Depth](#)
- typed clients, [Typed Clients](#)-Typed client created via client-gen
- TypeMeta, [TypeMeta](#)

U

- UNIX tooling, for packaging, [Other Packaging Options](#)
- user agents, [Client Options](#)

V

- validating plug-ins, [Admission](#)
- validation, [Validation](#)
- vendoring
 - dep, [dep](#)
 - glide, [glide](#)
 - Go modules, [Go Modules](#)
 - role of, [Vendoring](#)
 - tools for, [Vendoring](#)
- version control, [Technology You Need to Understand](#)
- versioning
 - conversion webhook architecture, [Conversion Webhook Architecture](#)
 - conversion webhook deployment, [Deploying the Conversion Webhook](#)
 - conversion webhook implementation, [Conversion Webhook Implementation](#)
 - example, [Revising the Pizza Restaurant](#)
 - HTTPS server setup, [Setting Up the HTTPS Server](#)
 - overview of, [Custom Resource Versioning](#)
 - process of, [Seeing Conversion in Action](#)
- versions, in Kubernetes API, [API Terminology](#), [Versioning and Compatibility](#)

W

- WATCH verb, [The Client Library](#)
- watches, [Events](#), [Watches](#), [Client Options](#)

- webhooks
 - admission webhooks, [Admission Webhooks](#)-[Admission Webhook in Action](#)
 - conversion webhook architecture, [Conversion Webhook Architecture](#)
 - conversion webhook deployment, [Deploying the Conversion Webhook](#)
 - conversion webhook implementation, [Conversion Webhook Implementation](#)
- work queues, [The Control Loop](#), [Work Queue](#)
- write access, [Getting the Permissions Right](#)

X

- x-kubernetes-embedded-object: true, [IntOrString](#) and [RawExtensions](#)
- x-kubernetes-int-or-string: true, [IntOrString](#) and [RawExtensions](#)
- x-kubernetes-preserve-unknown-fields: true, [Controlling Pruning](#)

Y

- YAML manifests, [Packaging](#): The Challenge
- ytt, [Other Packaging Options](#)

Z

- Zalando's Kopf, [Other Approaches](#)

关于作者

Michael Hausenblas是亚马逊网络服务的开发者倡导者，亚马逊网络服务是容器服务团队的一部分，专注于集装箱安全。Michael通过演示，博客文章，书籍，公开演讲以及对开源软件的贡献，分享他在云原生基础设施和应用程序方面的经验。在AWS之前，Michael曾在Red Hat，Mesosphere，MapR以及爱尔兰和奥地利的两个研究机构工作。

Stefan Schimanski是Red Hat的Go，Kubernetes和OpenShift的首席软件工程师。他的重点是Kubernetes API服务器，特别是CustomResourceDefinitions，API Machinery的实现，以及Kubernetes staging存储库client-go，apimachinery，api等的发布。在Red Hat之前，Stefan在Mesosphere工作过Marathon，Spark及其Kubernetes产品，并担任高可用性和分布式系统的自由职业者和顾问。在以前的生活中，斯特凡在数学逻辑方面研究过建构性数学，类型系统和lambda演算。

版本说明

*Programming Kubernetes*封面上的动物是绿鹳（*Tringa ochropus*）。属和物种名称均来自古希腊语。一只名为*trungas*的小型涉水鸟曾引起亚里士多德的注意，并且*ochropus*分解为古希腊语中的“赭石”和“脚”，*okhros*和*pous*。

绿鹳只有一个亲密的生物亲戚：孤独的鹳。绿色矶鹬享有极其广泛的范围，几乎遍及每个大陆。它们原产于亚洲，并在冬季迁移到温暖的气候。它们涉及各种沼泽环境并进食。在池塘，河流和潮湿的林地中，绿色鹳可以吃昆虫，蜘蛛，小甲壳类动物，鱼类和植物。

绿色鹳有宽阔的乳房和短颈。他们的喙长而纤细。近距离，它们的绿褐色翅膀显示出小而轻的点。这种羽毛着色与它们的蛋相反，它们具有棕色斑点的浅黄色。一个典型的离合器平均有两到四个蛋，在三周内孵化。绿鹳在其他鸟类甚至松鼠的废弃巢穴中孵化。

O'Reilly封面上的许多动物都濒临灭绝；所有这些对世界都很重要。

封面插图由Karen Montgomery设计，基于*Shaw's Zoology*的黑白雕刻。封面字体是Gilroy Semibold和Guardian Sans。文本字体是Adobe Minion Pro；标题字体是Adobe Myriad Condensed；代码字体是 Dalton Maag 的 Ubuntu Mono。