分类号 0175.2 密级 限制

UDC 004.72

学 位 论 文

声明式的通用 Kubernetes Operator 的设计与实现

（题名和副题名）

（作者姓名）

指导教师姓名、职务、职称、学位、单位名称及地址

南京大学计算机科学与技术系 南京市栖霞区仙林大道 163 号 210023

申请学位级别 硕士 专业名称

论文提交日期 2021 年 4 月 15 日 论文答辩日期 2021 年 6 月 1 日

学位授予单位和日期

答辩委员会主席： 教授

评阅人： 教授

副教授

教授

研究员

南京大學

研 究 生 毕 业 论 文

（申请硕士学位）

论 文 题 目 声明式的通用 **Kubernetes Operator** 的设计与实现

作 者 姓 名

学科、专业方向

研 究 方 向

指 导 教 师

2021 年 **5** 月 **14** 日

学　　号:

论文答辩日期: **2021** 年 **6** 月 **1** 日

指 导 教 师:　　（签字）

The Design and Implementation of A Kubernetes Operator Which is Declarative and Universal

by

Supervised by

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
MASTER
in

Department of Computer Science and Technology
Nanjing University

Apr 15, 2021

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：声明式的通用 Kubernetes Operator 的设计与实现

专业 XXXX 级硕士生姓名：

指导教师（姓名、职称）：

摘　要

Kubernetes 是最受欢迎的容器编排系统，它可以实现应用的自动化部署，已经成为分布式资源调度和自动化运维的事实标准。为了适应成千上万的应用的工作模式，Kubernetes Operators 被官方推荐作为在 Kubernetes 中打包、部署和管理应用的方法，它是用户扩展 Kubernetes 最主流的方式。现在开源社区已经诞生了很多优质的 Operators，帮助 Kubernetes 用户简化了很多应用的部署和管理，也为开发者提供了宝贵的参考。

Kubernetes 通过声明式 API 实现自身的易用性，用户只需要描述自己期望的状态而不用去考虑如何操作达到该状态。Operators 被用于扩展 Kubernetes 的声明式 API，在自定义控制器中实现用户自定义的行为。本文工作针对 Kubernetes Operator 开发中存在的学习曲线陡峭、非功能性代码繁多、模版代码冗余等问题，研究了 Operator 的工作原理，提出了一种声明式的通用 Kubernetes Operator，将其命名为 UniversalController，简称 UC。UniversalController 帮助开发者免除学习 Kubernetes 客户端库、Kubernetes API 机制库或其他工具的负担，也不用去编写或生成模版代码，而是将精力集中在业务逻辑上，最后使用 UniversalController 扩展的声明式 API 向系统注册即可实现自己的 Operator。而且 UniversalController 是语言无关的，开发者可以用自己熟悉或擅长的语言去实现所需的 Operator。

本工作目前已经基本开发完成，实现了必要功能，在后续完善阶段。已经基于 UniversalController 重构了若干个 Operators，包括官方示例 sample-controller、原生资源 StatefulSet、用于发布 tesorflow 深度学习任务的 tfjobs operator 等，验证了 UniversalController 的有效性和通用性。

关键词： Kubernetes；Operator；声明式；通用

南京大学研究生毕业论文英文摘要首页用纸

THESIS: The Design and Implementation of A Kubernetes Operator Which is Declarative and Universal

SPECIALIZATION:

POSTGRADUATE:

MENTOR:

Abstract

Kubernetes is the most popular container orchestration system for automating application deployment and has become the fact standard for distributed resource scheduling and automated operations and maintenance. To accommodate the working patterns of thousands of applications, Kubernetes Operators are officially recommended as the way to package, deploy and manage applications in Kubernetes, and it is the most mainstream way for users to scale Kubernetes. The open source community has now given birth to many high-quality Operators that help Kubernetes users simplify the deployment and management of many applications, and provide a valuable reference for developers.

Operators are used to extend the declarative API of Kubernetes to implement user-defined behavior in custom controllers. The work in this paper addresses the problems of Kubernetes Operator development, such as steep learning curve, extensive non-functional code, and redundant template code. This paper investigates how Operators work, and proposes a declarative universal Kubernetes Operator, which is named UniversalController, or UC for short. UniversalController helps developers eliminate the burden of learning Kubernetes client libraries, Kubernetes API mechanism libraries, or other tools, and instead of writing or generating template code, they can focus on business logic and finally use the UniversalController to extend the declarative API to register with the system to implement their own applications. UniversalController is language agnostic, so developers can implement their own operators in the languages they know or are good at.

This work has been developed completely and implemented the necessary features, in the subsequent refinement phase. Several Operators have been refactored based on

UniversalController, including the official sample-controller, the native resource StatefulSet, the tfjobs operator for publishing tesorflow deep learning tasks, etc. They are used to verify the The effectiveness and versatility of UniversalController.

keywords: Kubernetes; Operator; declarative; Universal

目　　次

插图清单

附表清单

第一章　绪论

1.1　研究背景

过去十年是云计算告诉发展的一个时间段，这期间新技术不断演进、优秀开源项目大量涌现. 通过树立技术标准与构建开发者生态，开源将云计算实施逐渐标准化。Docker 的出现使容器镜像迅速成为了应用分发的工业标准。随后开源的 Kubernetes，凭借优秀的开放性、可扩展性以及活跃开发者社区，在容器编排之战中脱颖而出，成为分布式资源调度和自动化运维的事实标准。这些技术的出现旨在将云应用中的非业务代码部分进行最大化的剥离，从而让云设施接管应用中原有的大量非功能特性，使业务不再有非功能性业务中断困扰的同时，具备轻量、敏捷、高度自动化的特点。

Kubernetes 具有很好的开放性与可拓展性，开发人员可以通过 Operator 来拓展 Kubernetes 的声明式 API，这也是最常用的方式。Operator 的概念是由 coreOS 提出的，是对 Kubernetes 的软件拓展，帮助实现应用程序的自动化部署、升级、管理以及运维。然而，编写一个 Operator 并不容易，具有相当高的门槛，并且需要付出大量的精力和时间。Operator 开发人员需要一定程度的 Kubernetes 和分布式系统知识，需要写大量的模版代码或者使用代码生成工具，编写出的 Operator 帮助我们实现了应用程序的自动化运维，但是维护这个 Operator 却还是要给开发人员带来很大的负担。因此诞生了很多工具，它们都希望帮助开发人员更简单的实现自己的 Operator。本文提出的 UniversalController 是一个声明式的通用 Operator，可以有效减少开发人员的开发与运维负担。

1.2　研究现状

为了简化 Kubernetes Operator 的开发，目前的方法主要是使用 SDK 工具，它使用代码生成工具来生成模版代码，帮助用户搭建项目基础脚手架。

Kubernetes-sigs 团队开源的 kubebuilder 和 coreOS 开源的 Operator SDK 都

是基于这个思路而产生的。与 Ruby on Rails 和 SpringBoot 等 Web 开发框架类似，Kubebuilder 和 Operator SDK 提高了开发人员在 Go 中快速构建和发布 Kubernetes API 的速度并降低了管理的复杂性。它建立在用于构建核心 Kubernetes API 的规范技术之上，以提供简单的抽象，减少模板和编码量。它们减轻了工作量，定义了一套自己的编程规范，不按照规范走就无法使用代码生成工具。但是它们的版本兼容性存在问题，新版本的编程规范会与就版本冲突，导致升级后无法使用，必须手动修改相关代码实现迁移。而且它们生成的代码依然是用 Go 编写的，整个项目依然是一个 Go 项目，用户依然需要具备 Go 语言和 Kubernetes 相关依赖库的基础知识。

1.3 本文工作

本文提出了一种声明式的通用 Kubernetes Operator，为用户开发 Operator 提供一种简单的新方式，让用户摆脱 Go 语言、Kubernetes 开发工具包、代码生成工具的学习与使用成本，用更加声明式的方式开发 Operator，将注意力完全集中在核心业务逻辑上，并且可以使用任意自己喜欢或熟悉的语言来实现一个标准优质的 Operator。本文将该工具成为 UniversalController，它自身也是一个 Operator，底层实现是经典的控制器模式，但是把业务逻辑部分抽取出来托管给用户编写的 hooks。

借助 UniversalController 提供的声明式 API，用户在写核心业务逻辑时也可以获得平时使用 yaml 编写配置文件并使用 kubectl apply 部署相近的体验，只是需要改用 json 编写一些配置文件。如果用户已经很熟悉用 kubectl apply 去使用 Kubernetes 的声明式 API 来管理应用，那么就可以很容易地基于 UniversalController 实现一个 Operator 为应用的部署、更新、维护提供自动化流程而不必去学习 Go 语言或者如何使用 Kubernetes 客户端库，也不需要去学习使用代码生成工具。

1.4 论文结构

本文共六章，组织结构如下：

第 1章绪论。本章对 Kubernetes 的流行程度，Kubernetes Operator 在其中扮演的角色和意义、现阶段开发 Operator 存在的问题做了简单的介绍。

第2章相关工作和技术。本章主要介绍 Operator 开发所涉及到的关键技术与工作。

第3章通用 Kubernetes Operator 需求分析与设计。本章首先对通用的 Kubernetes Operator 进行了需求分析。然后对 UniversalController 进行了设计，介绍了总体架构和各个模块。

第4章通用 Kubernetes Operator 实现。本章在第3章设计的基础上描述本文提出的 UniversalController 的具体实现。

第5章实验评估。本章介绍通过 UniversalController 实现的若干个 Operators，并对它们分别进行测试，验证 UniversalController 的易用性和有效性。

第6章总结和展望。总结本文所做的工作，并对 UniversalController 的未来发展做出进一步展望。

第二章　相关工作和技术

2.1　容器虚拟化技术

操作系统级的虚拟化，也被称为容器化，是指操作系统的一种功能，其中内核允许存在多个孤立的用户空间实例，这些事例被称为容器。

在容器内运行的程序只能看到容器的资源，即连接的设备，也就是卷；文件和文件夹；网络；容器的操作系统和架构；CPU 和内存。看起来容器类似于虚拟机，但是，与虚拟机不同的是，容器化允许应用程序使用与它们运行的系统相同的 Linux 内核，而不是创建一个完整的虚拟操作系统。在类 Unix 操作系统上，这个功能可以看作是标准 chroot 机制的高级实现，它改变了当前运行进程及其子进程的表象根文件夹。

操作系统级的虚拟化在最近几年开始流行，但却是一个老概念。chroot 机制是在 1979 年第七版 Unix 中发布的。然后，这个功能在 FreeBSD Jails 中得到了扩展，在 2000 年 3 月发布的 FreeBSD v4.0 中引入。在接下来的 9 年时间里，FreeBSD Jails 增加了 CPU 和内存限制、磁盘空间和文件数量限制、进程限制以及多 IP 的网络等功能。2001 年，Linux-VServer 被发布用来创建 VPS（虚拟私人服务器）和隔离的虚拟主机空间。

在 2013 年美国 PyCon 大会上，Solomon Hykes 提出了 Docker 是容器的未来。Docker 是一个基于 Linux 的平台，通过基于容器的虚拟化来开发、运输和运行应用程序。和前面介绍的系统一样，它利用主机的操作系统内核来运行多个隔离的用户空间实例，在 Docker 中这些实例被称为容器。Docker 发展很快，迅速成为云容器化的事实标准[3]。

在 Docker 发布仅 7 个月后，Docker 和红帽宣布重大合作，包括兼容 Fedora/RHEL，并开始在红帽 OpenShift 内使用 Docker 作为容器标准[4]。次年 Kubernetes 诞生，并在 2015 年被采用为完全重新设计的 OpenShift 3.0 的基础[5]。

2.1.1 Docker 容器工作方式

每个 Docker 容器都打包了要运行的应用程序，以及它们所需要的任何软件支持（例如，库、二进制文件等）。这是通过将所有的东西打包在一个 Docker 镜像中，然后可以实例化获得 Docker 容器。Docker 镜像是由层组成的。一个镜像可以从一个空的镜像（一个被称为"scratch" 的特殊镜像）创建，也可以在其他现有镜像之上创建。例如，我们可以以 Ubuntu 镜像为基础，把文件复制进去，或者执行 apt install 等命令。每一个在镜像上运行的操作都会创建一个新的层。这样就可以在镜像之间重用通用层，节省空间。现有的 Docker 镜像是通过 Docker registries 发布的，比如 Docker Hub 是官方的 registry，Quay 是 OpenShift 的 registry，同时还存在其他私有或公共 registry。

Docker 容器是不持久的，当容器停止时，由容器产生的数据默认会丢失。这就是 Docker 引入卷的原因，卷是专门指定的目录（在一个或多个容器内），其目的是持久化数据，独立于挂载了它们的容器的生命周期。Docker 从来不会在容器被移除时自动删除卷，也不会移除不再被任何容器引用的卷。Kubernetes 和 OpenShift 都允许拥有分布式卷，将存储从计算部分分离出来。

Docker 允许容器间相互通信。它允许创建虚拟网络，从简单桥接网络（针对单个主机），到复杂的覆盖网络（针对主机集群）。例如在 Kubernetes 中，有分布在许多节点上的微服务之间的网络，允许应用程序透明地进行相互通信。

2.2 Kubernetes

Kubernetes 是一个开源系统，用于自动化部署、扩展和管理容器化应用[6]。它负责编排组成应用的容器，方便应用的管理和发现。

Kubernetes 的诞生源于 Google 内部的容器编排工具 Borg，Google 在 2014 年 6 月[7] 将其开源[8]。虽然 Borg 是用 C++ 编写的，但 Kubernetes 是用 Go 实现的，Go 是谷歌设计的一种静态类型化、编译的编程语言。因此，Go 是大部分 Kubernetes 相关项目的语言，包括 Operators。

2.2.1 Kubernetes 集群架构

即使 Kubernetes 可以运行在单个物理节点上，例如 Kubernetes Minikube 和 OpenShift CloudReady Containers 允许在本地运行单节点 Kubernetes，但它通常是运行在多个主机的集群上。

图 2-1所示为 Kubernetes 架构，一个 Kubernetes 集群由若干个节点组成，在这些节点中有一个或多个主节点，在主节点上运行着 Kubernetes 的控制面组件。

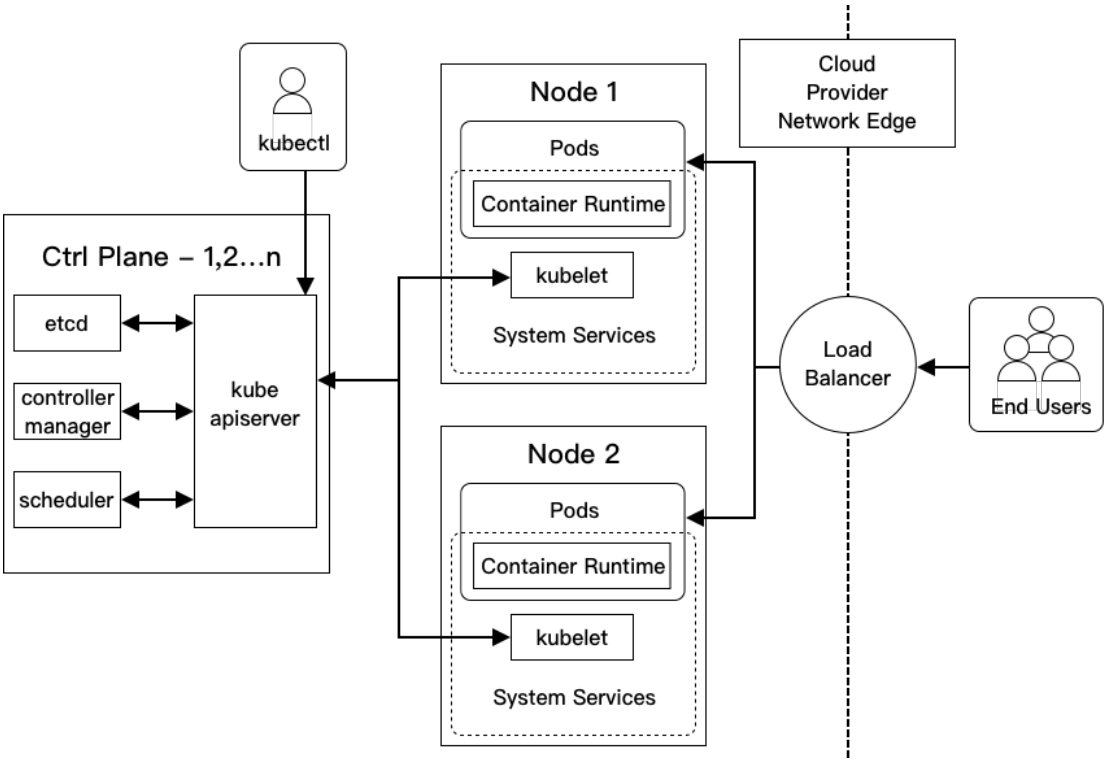

图 2-1: Kubernetes 架构图

节点

Kubernetes 集群中的主机称为节点，可以是虚拟机，也可以是物理机。一个节点有以下信息[9]。

- **Address**，由主机名、机器的外部 IP 和集群中的内部 IP 组成。
- **Capacity 和 Allocatable**，描述了节点的 CPU、内存和存储边界，以及可以在其上部署多少进程。
- **Conditions**，例如“Ready”状态，“NetworkUnavailable 状态”以及一些关于

节点压力的信息：“MemoryPressure”会对内存占用率过高的情况发出警告；“DiskPressure”会对硬盘占用率过高的情况发出警告；“PIDPressure”会对该节点上分配的进程过多的情况发出警告。这些标志会根据节点当前的实际状态进行更新。

- **Info**，描述了节点的一般信息，如内核版本、Kubernetes 版本、Docker 版本（如果使用）和操作系统名称。

主节点

主节点充当集群的网关和大脑，将 API 暴露给用户使用从而与 Kubernetes 交互。主节点是 Kubernetes 集群的入口，负责 Kubernetes 提供的大部分中心化逻辑。它决定了如何调度任务，即如何将工作分割和分配给节点。它还收集节点的日志和健康状态。主节点通常专门负责编排容器，而节点则负责实际的运行容器。

在高可用性集群中，多个主节点被实例化，以实现冗余备份，目的是在低于一定数量的主节点故障的情况下也能保证集群的运行[10]。

2.2.2 Kubernetes 控制面

如图 2-2所示，Kubernetes 控制面由运行在主节点上的各种组件组成。

etcd

etcd 对于 Kubernetes 跨节点工作至关重要，因为它提供了一个轻量级和分布式的键值存储，可以跨越多个节点。Kubernetes 使用 etcd 来存储配置数据，这些数据可以被集群中的每个节点访问。它通常安装在主节点上，在生产系统和高可用性集群中，安装在多个主节点上，以实现冗余和弹性。

kube-apiserver

kube-apiserver 是整个集群的主要控制模块。所有的管理工具，包括 Kubernetes 命令行工具 kubectl，都是通过它暴露的那些 API 与 Kubernetes 进行通信的。kubectl 是默认的从本地计算机与 Kubernetes 集群交互的方法，允许管理集群、部署及管理 Kubernetes 对象。

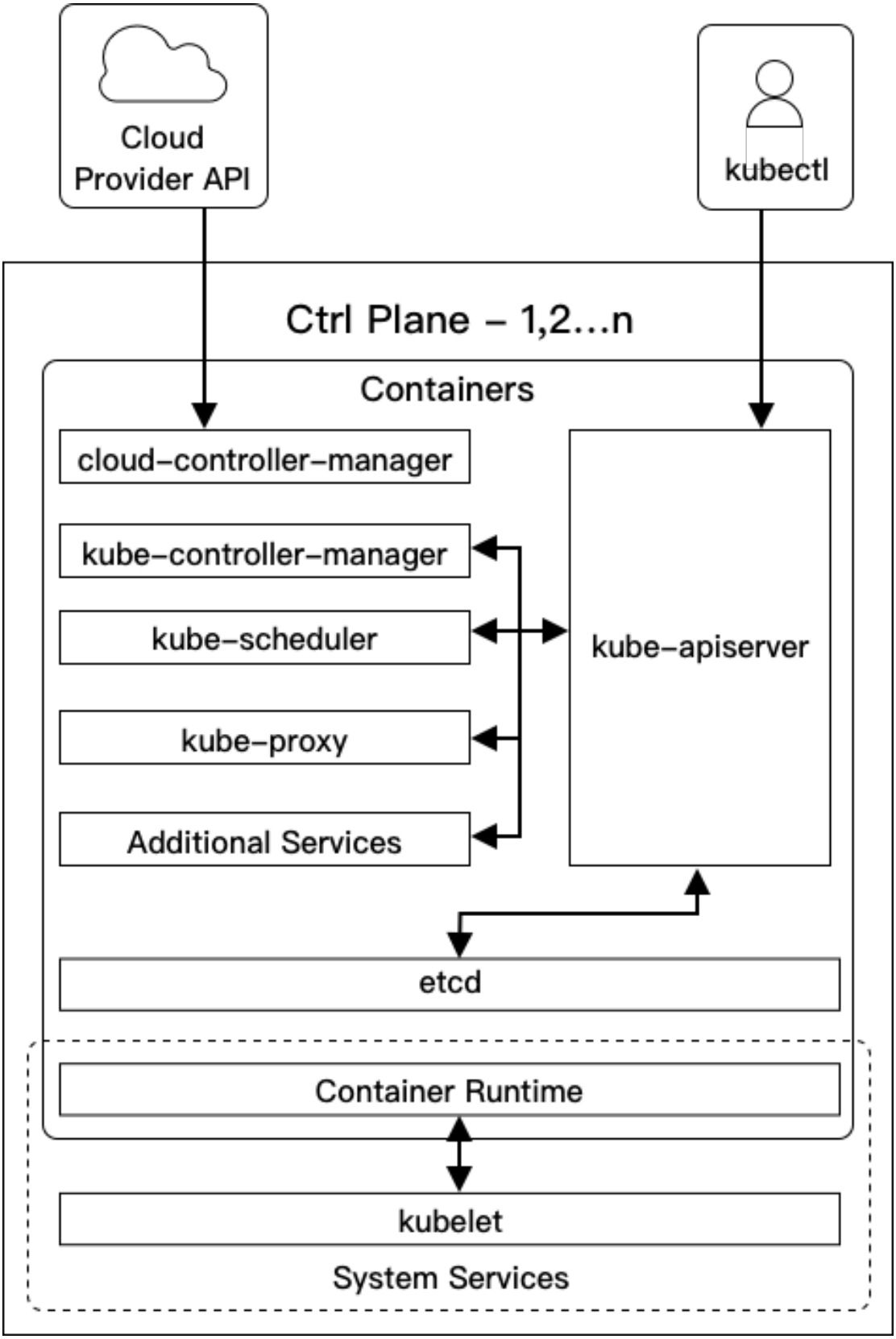

图 2–2: Kubernetes 控制面[1]

kube-scheduler

kube-scheduler 是一个通过分析当前基础设施环境将工作负载分配给节点的服务。为此，它需要跟踪每台主机上的可用容量，以确保工作负载不会超过某个节点或整个集群上的可用资源[11]。

kube-controller-manager

kube-controller-manager 是一个具有许多职责的通用服务，可以将其视为控制器组件的集合。这其中的每一个控制器都会调节集群的状态，管理工作负载生命周期，或者执行常规任务[1]。

当检测到一个变化时，控制器读取新的信息并执行满足所需状态的程序。这可能涉及到扩大或缩小应用程序的规模，调整端点等。例如，ReplicaSet 确保为一个应用程序定义的副本数量与当前部署在集群上的数量相匹配。

cloud-controller-manager

Kubernetes 可以部署在许多不同的环境中，例如，裸机服务器、AWS（Amazon Web Services）、GKE（Google Kubernetes Engine）、Azure 等等。每一种服务都有不同的 API 和 Kubernetes 对象的实现。Kubernetes 必须与这各种各样的基础设施和云提供商进行交互，将它们的非同质资源映射到其资源抽象中。cloud-controller-manager 作为 Kubernetes 和底层基础设施之间的粘合剂。它告诉 Kubernetes 如何与目标基础设施的不同能力、特性和 API 进行交互。由于一个 Kubernetes 可能同时分布在多个环境中，所以在一个集群中可以有多个 cloud-controller-manager，每个环境都有一个。

2.2.3 节点服务组件

Kubernetes 控制面只需要部署在主服务器上，同时一些组件也必须安装在每个节点上，图 2-3展示了这些组件。

容器运行时

容器运行时负责启动和管理容器。Docker 是典型的容器运行时，同时还有很多其他的选择，例如 podman、containerd 等，云提供商也可以提供自己定制的容器运行时来满足这一组件要求。

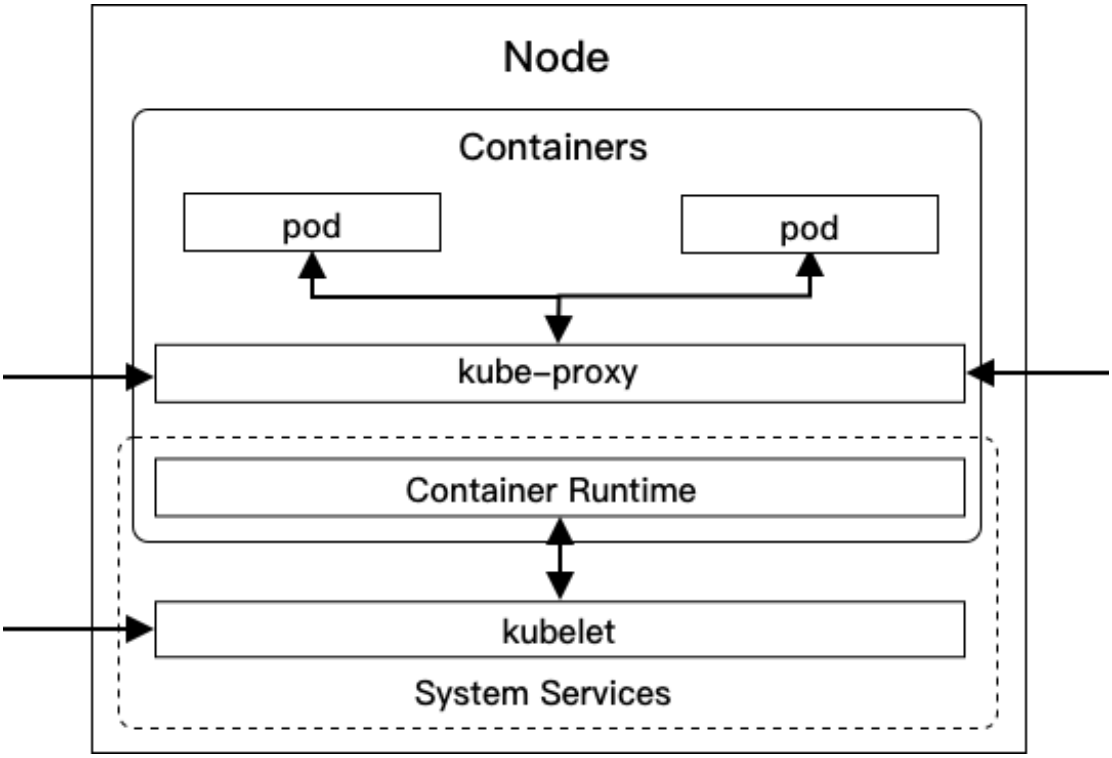

图 2-3: Kubernetes 节点服务组件[1]

kube-proxy

kube-proxy 负责在节点之间创建网络，从而让节点加入 Kubernetes 集群接收流量。它还将传入节点的请求转发到正确的容器，也实现了一些基本形式的负载均衡。另外，它确保每个网络环境的正确隔离。

kubelet

Kubelet 负责与控制面服务交换信息。它从主节点组件接收命令和任务。任务以 manifest 的形式接收，manifest 定义了要部署的资源和操作参数。在接收到一堆任务后，kubelet 会负责执行工作和维护节点服务器上相关资源，一切都通过与容器运行时的交互来完成。

2.2.4 Kubernetes 对象

Kubernetes 对象也被叫做资源（resources），Kubernetes 自身定义了一组构件，提供了部署、维护和扩展应用的机制，这些构件都是“Kubernetes native resources”。这些基元可以通过 Kubernetes API 进行扩展，基于这一点 Kubernetes

Operators 才可以实现。利用 Kubernetes 扩展性实现的工具可以作为 Kubernetes 容器运行，方便相关应用的部署与管理。

Spec 和 Status

几乎每一个 Kubernetes 对象都包括两个嵌套的对象字段，用来管理该对象的配置，即对象规格（spec）和对象状态（status）。规格描述了对象的期望状态，而其状态则描述了当前的状态。Kubernetes 不断地对对象进行操作，目标是最终让当前状态与 spec 中给出的期望状态相匹配。

Pod

Pod 是 Kubernetes 中的基本部署单元，也是最小单元。一个 Pod 就是一群鲸鱼的社会群体[9]。这个比喻很形象：Kubernetes Pod 是一组容器化组件的抽象。一个或多个紧密耦合的容器被封装在一个对象中，保证它们在同一台主机上被共同定位和调度，并且可以共享资源。

Pod 中的容器紧密地运行在一起，共享一个生命周期，并且总是被调度到一个节点上。Kubernetes 将它们作为一个单元进行管理，它们共享环境、卷和 IP 空间。Pod 中的所有容器都可以在 localhost 上相互引用，正因为如此，Pod 中的应用必须协调它们对端口的使用。一个容器的暴露端口也会从 Pod 中暴露出来。从容器外部，无法与 Pod 中的某个容器进行通信，我们只能与 Pod 进行通信，从而到达相应端口的容器。

像容器一样，Pods 被认为是相对短暂的实体，而不是持久的。Pod 的唯一标识符 (UID) 和 IP 会一直保持到 Pod 终止。终止可能发生在用户发送终止命令时（有宽限期），宽限期结束时，新的版本被部署，旧的 Pod 在过渡期后被认为是死亡的。当 Pod 内的命令终止时，Pod 也会死亡[9]。

通常情况下，Pods 由一个满足工作目的的主容器和一些帮助容器（称为 sidecars）组成，这些容器可以帮助完成密切相关的任务。这些程序在自己的容器中运行，但与主应用程序紧密相连[11]。例如，一个要在后台运行的周期性任务，如图 2-4中的“File Puller”，它负责与远端同步数据，而“Web Server”作为主服务器启动一个 Web 服务供用户访问查看这些数据，他们使用同一个 volume，从而共享其中存储的文件。

因此，在一个 Pod 中运行多个容器与在一个容器中运行多个程序是完全不同的，即可以达到在 localhost 上拥有多个程序的所有优点，同时获得透明度、

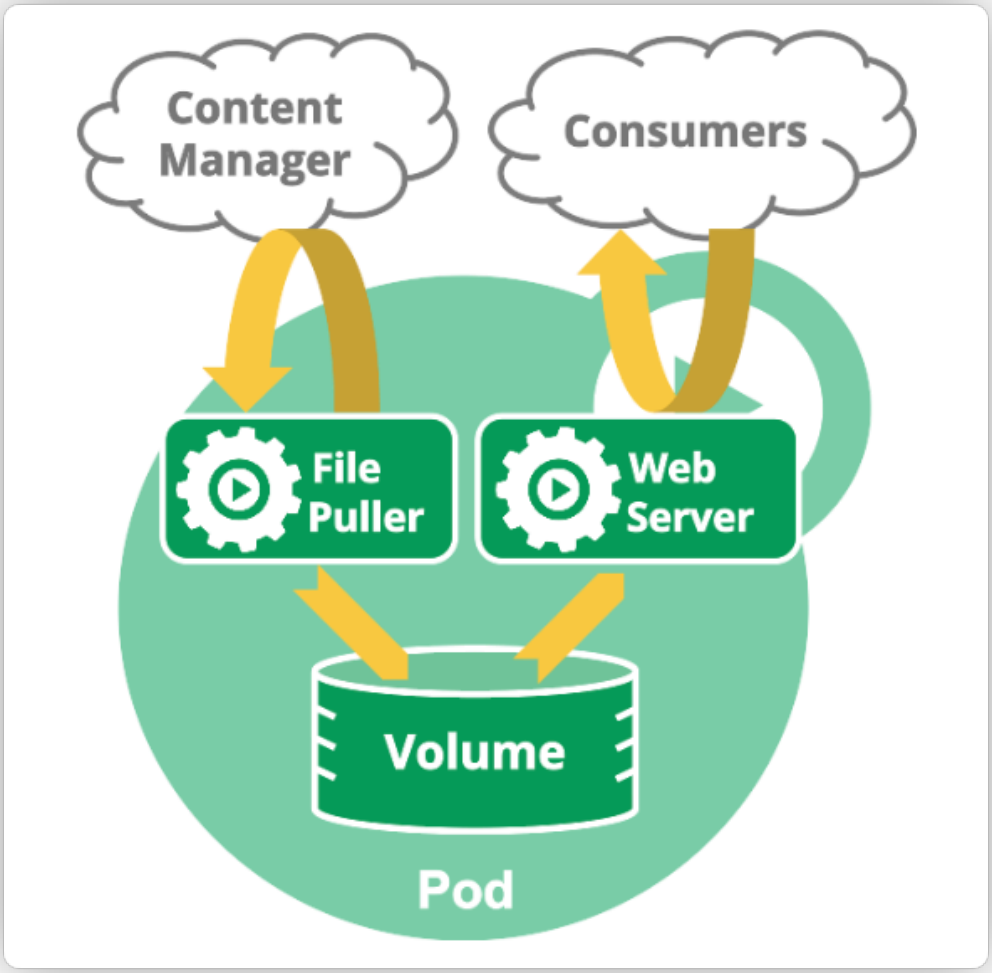

图 2-4: Pod 示例

软件依赖性的解耦、效率和易用性。水平缩放不应该在 Pod 内部通过容器增减实现，而应该在外部，缩放 Pod[9]。

Volume

Kubernetes Volume 是存储的一个抽象。容器中的磁盘文件是短暂的，因为它们与容器一起死亡，当一个新的容器被部署时，它将从原始映像的干净状态开始。这可能会对有状态的应用程序造成问题，例如，存储系统或数据库。此外，有时在 Pod 中一起运行的容器需要共享文件。Volume 的抽象解决了这两个问题。

Volume 也不是完全持久化的。当一个 Pod 被移除时，相关的 Volumes 也会

随之被销毁。Kubernetes Persistent Volumes 避免了这一问题，它是一种抽象化的机制，它可以抽象出更强大的存储，不与 Pod 的生命周期挂钩[9]。Persistent Volumes 是集群的独立存储资源，可以通过“PersistentVolumeClaim”连接到 Pod 上。如果 Pod 被移除，持久化卷就会被释放而不会被删除。这样就可以将之前的存储重新连接到新版本的 Pod 上，恢复其数据。

ReplicaSet

ReplicaSet 的目的是维持一组稳定的副本 Pod 的正确运行。因此，它经常被用来保证指定数量的相同 Pod 的可用性[9]。ReplicaSet 通过根据需要创建和删除 Pod 来实现其目的，以达到期望的副本数量。当 ReplicaSet 需要创建新的 Pod 时，它使用 ReplicaSet spec 中的 Pod 模板来配置 Pod。

ReplicaSet 部署的所有 Pod 都会通过“ownerReference”字段链接到它。这帮助 ReplicaSet 知道它正在维护的 Pod 的数量和种类，从而去决定应该创建一个新的 Pod，还是删除其中一个现有的 Pod。有一种情况是一个 Pod 没有所有者参考，可能是因为它是用户手动创建的，但它符合 ReplicaSet 的选择器，它将立即被获取，设置“owerReference”字段为该 ReplicaSet。

Deployment

Deployment 为 Pods 和 ReplicaSets 提供声明式更新[9]。Deployment 是一种高级别对象，旨在简化副本 pods 的生命周期管理。一个 Deployment 描述了期望的状态，Deployment Controller 以可控的速度将实际状态改变为期望状态。Deployment 可以修改目标配置，Kubernetes 会调整 ReplicaSet，管理不同应用版本之间的过渡。

Service

如图 2-5所示，Service 是一种将运行在一组 Pod 上的应用作为网络服务公开的抽象方式[9]。一个 Kubernetes Service 也可以作为 pods 的内部负载平衡器。它将执行相同功能的 pods 收集在一起作为一个逻辑集合，使其呈现为一个单一实体，并在它们之间路由传入流量。一个 Pod 是否应该加入 Service 的集合由标签选择器定义。

Service 为非持久化的对象提供了一个稳定的端点。Service 允许根据需要扩展或替换后端单元。同时，无论它路由到的 Pod 集合如何变化，Service 的 IP

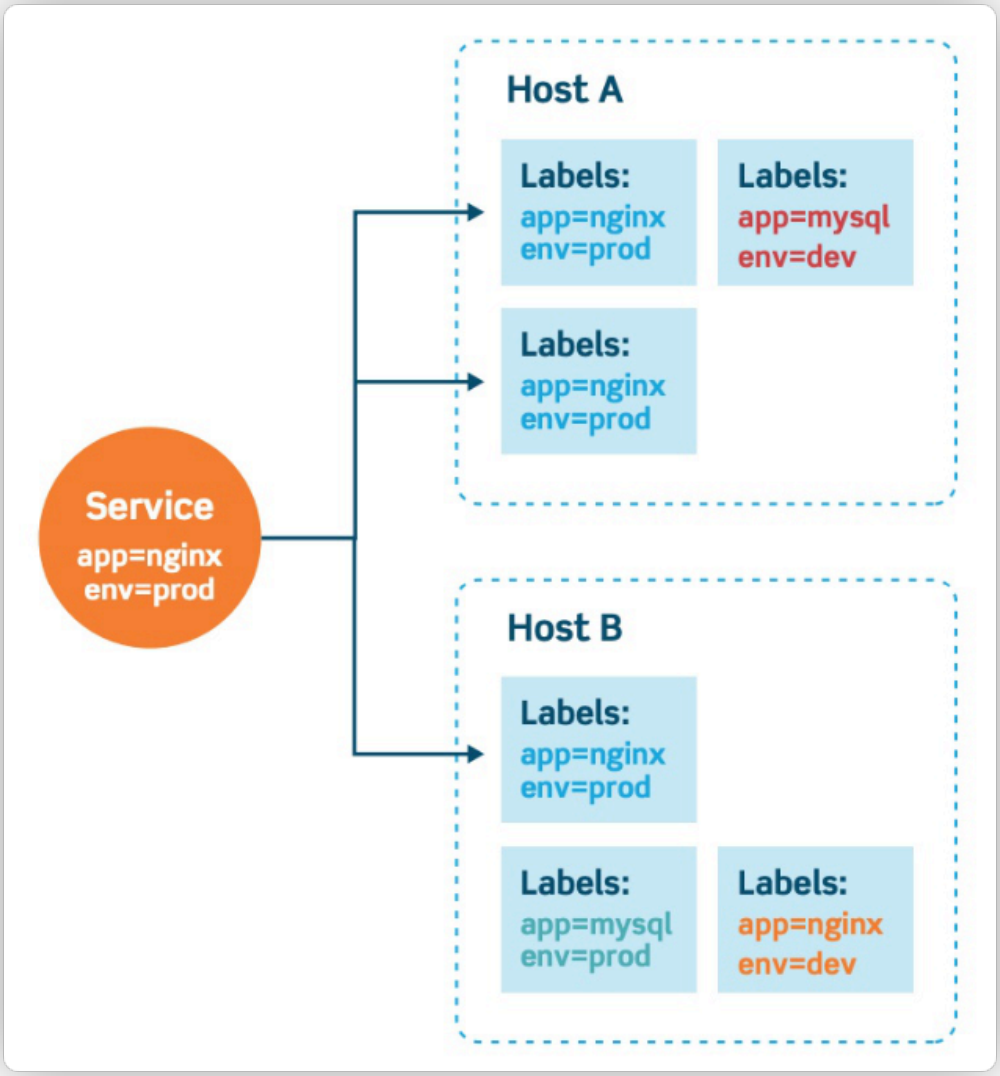

图 2-5: Service 标签示例[1]

地址都会保持稳定[11]。通过部署 Service，应用可以轻松获得可发现性，并可以简化容器设计。

Service 资源会收到一个 ClusterIP，仅在内部 IP 上暴露服务[1]。这使得服务只能从集群内部到达。NodePort 在每个节点的 IP 上通过指定的端口暴露服务。这个功能允许设置一个外部负载平衡器，或者在需要的时候，在其特定的端口上暴露应该从外部世界可达的服务。更多的时候，NodePorts 用于调试目的，以快速的方式暴露服务，避免在开发的第一阶段配置路由或 Ingresses。

ConfigMap

ConfigMap 也是一个 Kubernetes 对象，用于在键值对中存储非机密数据。Pods 可以将 ConfigMaps 作为环境变量、命令行参数或卷中的配置文件来使用。ConfigMaps 允许将特定环境的配置从容器镜像中解耦出来，从而使应用程序易于移植。

2.3 Kubernetes Operators

Kubernetes 官方文档将 Operators 描述为“Kubernetes 的软件扩展，利用自定义资源来管理应用程序及其组件”[12]。一个 Operator 本质上就是一个自定义资源类型和一个监视该资源类型并做出实际操作的自定义控制器的组合。控制器是 Kubernetes 中的一个核心概念，它被实现为一个控制循环，在 Kubernetes 中的一个 Pod 内持续运行，如图 2-6所示，它比较对象的期望状态和当前状态，并在需要时随时调和这种状态。事实上，当对象的当前状态与期望状态不同时，管理 Kubernetes Operator 会对对象发出指令，使其最终达到期望状态。

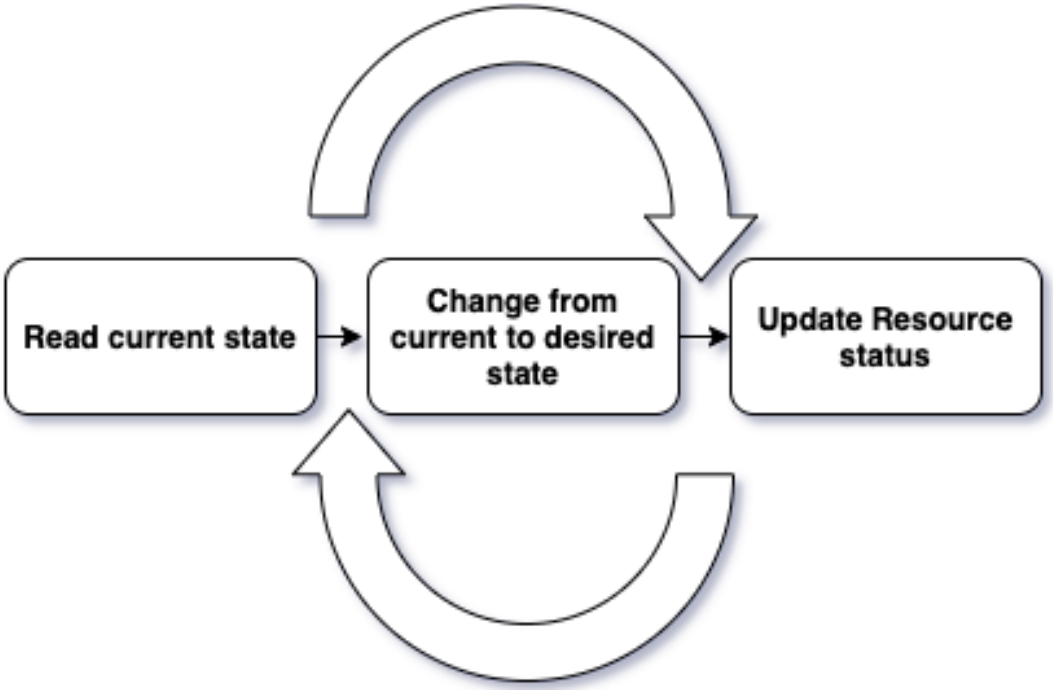

图 2-6: 控制循环

一个 Operator 是一个运行在集群上的 Pod 中的软件，与 Kubernetes API 服务器交互。它通过 CustomResourceDefinition（Kubernetes 中的一种扩展机制）

引入新的对象类型。Operator 监控这种自定义资源类型，并被通知其存在或修改。当 Operator 收到这种通知时，它将开始运行一次循环，以确保这些对象所代表的应用服务的所有所需连接实际上是可用的，并按照用户在对象 spec 中表达的方式进行配置。例如，Grafana Operator 将 Grafana 和 GrafanaDashboard 对象添加到 Kubernetes 中。希望部署 Grafana 的用户只需要创建一个 Grafana 对象，并将其所需配置作为参数填入自定义资源 spec 中。Operator 将部署所有必要的对象并保持其运行。

2.3.1 kubebuilder 与 Operator SDK

kubebuilder 和 Operator SDK 都是 Operator 的开发工具。kubebuilder 由 kubernetes 特别兴趣小组（sigs）推出，而 Operator SDK 由 CoreOS 推出，它们的功能非常接近，在稍早的版本中，它们生成的 Operator 项目框架和底层依赖会有些区别，但是现在已经达成合作，二者同质化，这里就一起介绍了。

它们都只为 Go 开发者服务，都相当依赖于代码生成器，都提供了一个命令行工具来创建一个新项目、生成代码、构建二进制文件、镜像、配置清单等，都把控制器模式作为一个库来实现。

它们的功能都很强大，但是缺点也很明显：

- 限定编程语言，提高了使用门槛，Go 语言有很多对新手不友好的地方。
- 本质上是代码生成工具，但是版本兼容性较差，不同的版本对应不同的 Kubernetes 版本，但是旧版本生成的项目必须手动升级成新版本项目，没有自动化工具。
- 用户依然要学习使用 Kubernetes 客户端库，用指令式代码与 Kubernetes 交互，编写旧版本资源向新资源迁移的代码，处理新旧资源的合并。

2.4 Serverless

随着以 Kubernetes 为代表的云原生技术成为云计算的容器界面,Kubernetes 成为云计算的新一代操作系统。面向特定领域的后端云服务 (BaaS) 则是这个操作系统上的服务 API, 存储、数据库、中间件、大数据、AI 等领域的大量产品与技术都开始提供全托管的云形态服务, 如今越来越多用户已习惯使用云服务, 而不是自己搭建存储系统、部署数据库软件。

当这些 BaaS 云服务日趋完善时,Serverless 因为屏蔽了服务器的各种运维复杂度,让开发人员可以将更多精力用于业务逻辑设计与实现,而逐渐成为云原生主流技术之一。Serverless 计算包含以下特征:

- **全托管的计算服务**，客户只需要编写代码构建应用,无需关注同质化的、负担繁重的基于服务器等基础设施的开发、运维、安全、高可用等工作。
- **通用性**，结合云 BaaS API 的能力,能够支撑云上所有重要类型的应用。
- **自动的弹性伸缩**，让用户无需为资源使用提前进行容量规划。
- **按量计费**，让企业使用成本得有效降低,无需为闲置资源付费。

函数计算 (Function as a Service) 是 Serverless 中最具代表性的产品形态。通过把应用逻辑拆分多个函数,每个函数都通过事件驱动的方式触发执行,例如当对象存储 (OSS) 中产生的上传 / 删除对象等事件,能够自动、可靠地触发 FaaS 函数处理且每个环节都是弹性和高可用的,客户能够快速实现大规模数据的实时并行处理。同样的,通过消息中间件和函数计算的集成,客户可以快速实现大规模消息的实时处理。

Serverless 计算与容器技术结合后诞生出了更多其他形式的服务形态。通过良好的可移植性,容器化的应用能够无差别地运行在开发机、自建机房以及公有云环境中; 基于容器工具链能够加快解决 Serverless 的交付。云厂商如阿里云提供了弹性容器实例 (ECI) 以及更上层的 Serverless 应用引擎 (SAE),Google 提供了 CloudRun 服务,这都帮助用户专注于容器化应用构建,而无需关心基础设施的管理成本。此外 Google 也开源了基于 Kubernetes 的 Serverless 应用框架 Knative。Bitnami 也开源了 kubeless，一个轻量易用的 serverless 框架。kubeless 是本文后续主要使用的工具之一，用于将用户编写的自定义调谐函数部署成一个 web 服务。

2.5 小结

本章主要介绍了 UniversalController 开发过程中所使用的相关技术以及使用技术的场景。首先介绍了容器化技术，这是如今云上应用的默认部署方式。然后介绍了容器编排系统 Kubernetes，它是实际的业界标准，接着介绍 Kubernetes Operator 这一 Kubernetes 上的重要概念与拓展方式，介绍了现有的开发工具已经开发中存在的问题。最后简要介绍了基于 UniversalController 开发 Operator 时会用到的 serverless 技术。

第三章　声明式的通用 Kubernetes Operator 的需求分析与设计

本文所陈述的声明式的通用 Kubernetes Operator，即 UniversalController，主要是为了让开发者更容易的去实现以及部署 Kubernetes Operators，进而扩展 Kubernetes 的 API。

3.1　需求分析

3.1.1　事件配置

首先，UniversalController 应该能够方便简明地通过配置指明调谐过程应该被那些事件源触发。在 Kubernetes 上实现自定义调谐的主要方法是将特定域的期望状态封装在一个或多个自定义资源中，由自定义控制器来监视并执行调谐。

出于这个原因，UniversalController 的监视配置功能应该允许用户初始化这个资源监视过程。为了最大限度地减少模板代码，配置接口应该尽量最小化，只需要特定于应用程序的必要参数，例如要监视的（自定义）资源的基本信息（apiVersion 和 kind）。

例如，开发人员在构建 TensorflowJob Operator 时，应该能够简单地指示 UniversalController 监视所有 TensorflowJob 类型的资源的事件。

3.1.2　支持动态资源类型

作为一个通用 Operator，UniversalController 事先不能知道它要处理哪些种类的资源。需要处理的是 K8s 内部定义的原生资源还是用户自定义资源，这些

在实现时都是未知数，只有在运行时才能知道，所以 UniversalController 需要支持动态资源类型，能够处理 Kubernetes 系统内任意一种资源。

3.1.3 易用的调谐接口

调谐逻辑是 Kubernetes Operators 的核心部分和价值，所以开发者还要能够轻松地提供以业务逻辑为中心的自定义调谐函数。

当实现调谐函数时，开发人员应该能够专注于自定义的业务逻辑，同时尽量减少需要编写和理解的底层 Kubernetes 或 UniversalController 特定逻辑。

因此，UniversalController 应该提供一个编程接口，用于接受自定义的调谐函数，它将与所提供的事件配置自动匹配。此外，为了让开发者编写代码时作出正确的决策，UniversalController 应该将上下文信息，如事件元数据，暴露给自定义代码。最后，开发者提供的代码还应该能够根据调谐的结果指导下一步的行动。

3.1.4 多控制器并存

当开发一些项目时，可能需要在一个进程中打包并部署多个自定义控制器，例如 kube-controller-manager 和 cloud-controller-manager 组件都是这样工作的。

在这种情况下，依然需要保持关注点的分离，使用具有不同配置的单独控制器来调谐不同的 Kubernetes Resources。但控制器的集合确实构成了一个便于协作的单元自动化服务，并且还可以共享一些 Kubernetes 特定的配置，如认证、订阅等。

因此，UniversalController 的另一个需求是支持这样的配置，允许定义和运行多个控制器。

3.1.5 降低工作量

图 3-1是一个 Operator 开发的基本流程。UniversalController 需要能够对此进行简化，舍去或简化一些步骤。就像一个 Operator 简化了某个应用的部署、更新与管理，UniversalController 需要能简化一个 controller 的开发、创建与管理。

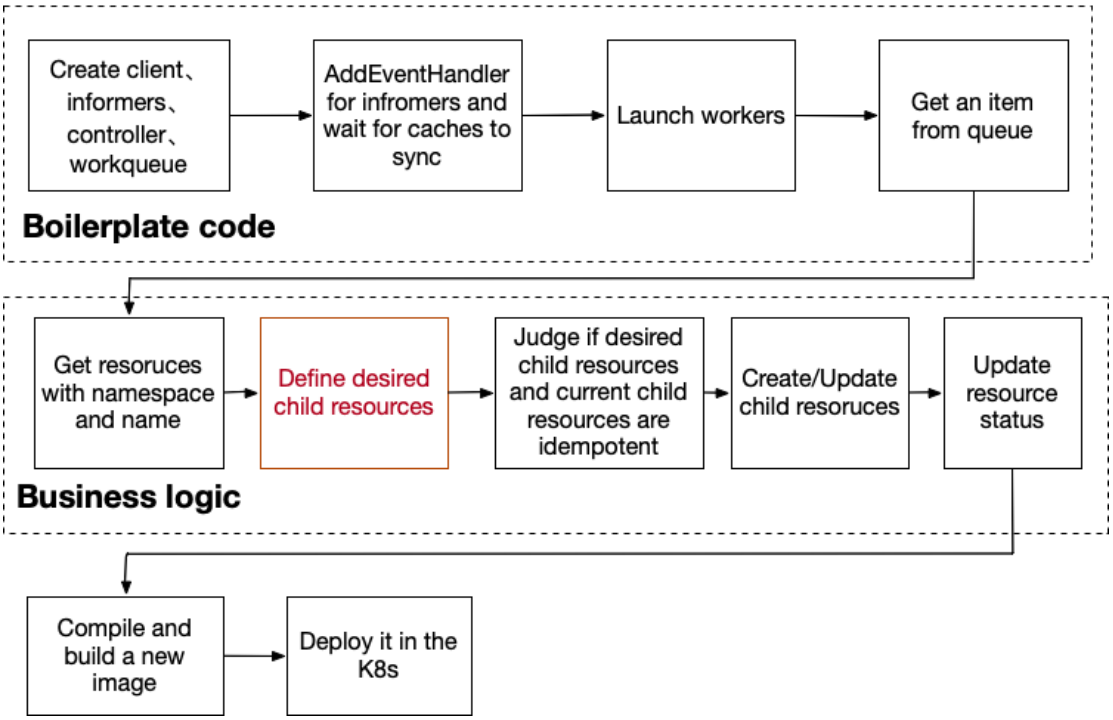

图 3-1: Operator 编写流程

3.1.6 语言无关

Kubernetes 本身是用 Go 语言开发的，因此官方也推荐用 Go 来开发 Operator，可以使与 Kubernetes APIs 的集成更简单。Kubernetes 的 Go 客户端 clieng-go 也确实是最古老且最成熟的，得到了完善的测试并且可靠，Kubernetes 组件内部也会使用它。而基于 client-go 的 controller-runtime 提供了更高级的抽象，让 Operator 的开发更加便利。但是其他编程语言则没有那么可靠成熟的 Operator 开发工具，比如 Python 和 Java 虽然有 Kubernetes 官方团队推出的客户端，但是都只提供了低级的抽象，用来实现 Operator 会显得极为繁琐。

针对这种现象，UniversalController 被设计成语言无关的，只要开发者使用的编程语言能够理解 json 就可以用来编写 Operator。

3.1.7 通用性

UniversalController 对一般的 Operator 做了一层高级的抽象，但是同时也需要保留充分的灵活性，以确保它能够适用于大部分情况，用户得到的应该是一种通用的开发方式。

3.2 设计

UniversalController 自身也是一个 Kubernetes Operator，负责监视 UniversalController CRD（下文简称 UC CRD）并维护它们对应的自定义控制器。创建一个新的 UC CRD 等同于在系统中注册一个新的自定义控制器。Kubernetes 提供的动态 API 注册机制 CustomResourceDefinition，这让用户可以声明式地创建一种新的自定义资源，UniversalController 扩展了 Kubernetes 的 API，UC CRD 接口进一步让用户可以声明式地创建自定义控制器。

3.2.1 总体架构

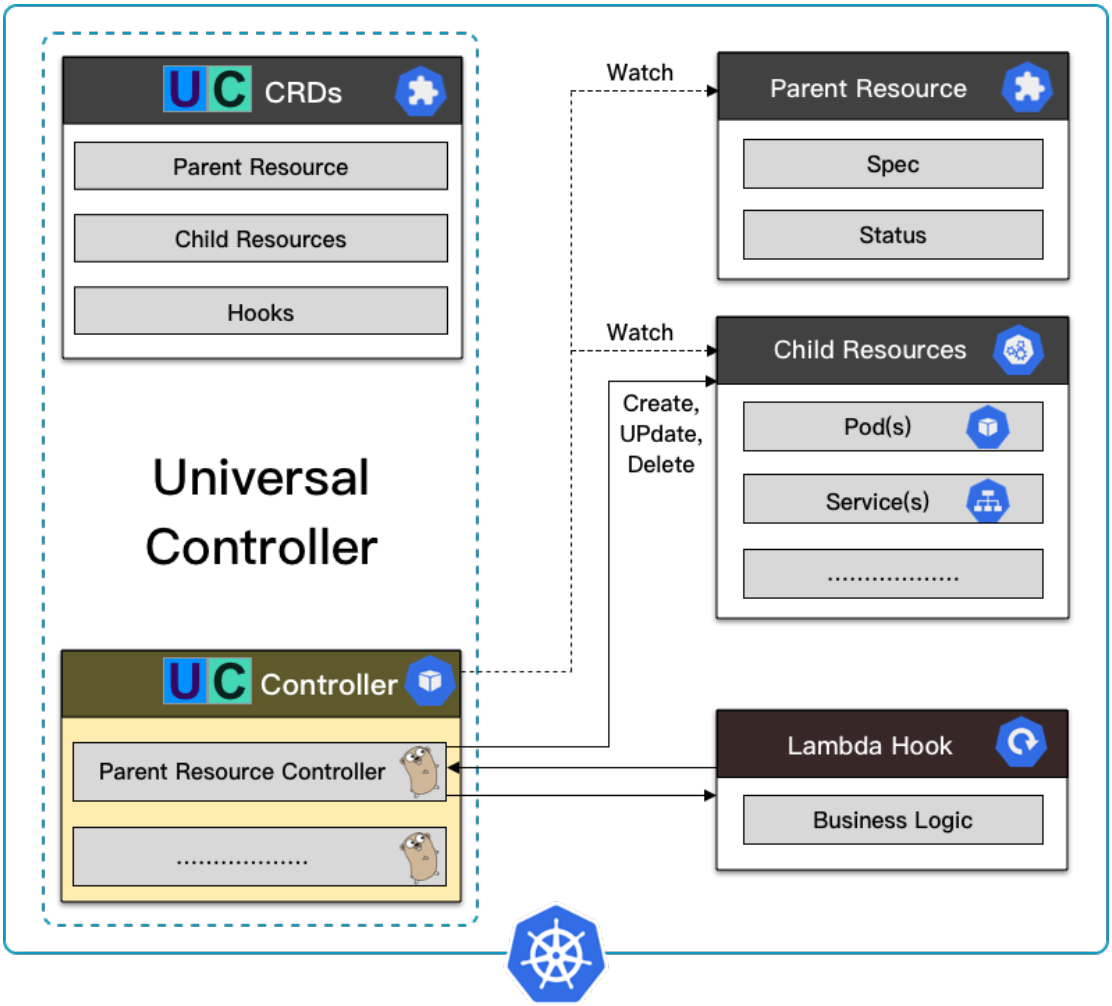

图 3-2: UniversalController 架构

图 3-2展示了 UniversalController 的总体架构。UniversalController 自身是一

个 Kubernetes 中的 controller，负责监视 UniversalController CRDs，当有一个新的 UniversalController CRD 被创建时，它会启动 Parent Resource 的控制器作为响应。也就是说，UniversalController 是控制器的控制器，可以用于管理多个控制器，实现多控制器并存在一个进程中运行。

3.2.2 自定义资源

UC CRD 是 UniversalController 提供的声明式 API，通过 Kubernetes CustomResourceDefinition 注册在 Kubernetes API 中。通过创建一个 UC 资源，可以很方便的定制一种控制器，它根据父对象中指定的期望状态管理一组子对象，这也是最常见的控制器类型。像 Deployment、StatefulSet、TFJob、PytorchJob 的控制器都是符合这种模式的。

UC CRD 是对控制器的高级抽象，包含了一个控制器运行时需要知道的各项信息，例如事件配置、更新策略和调谐接口访问地址。

- 用户只需要填写好 parentResource 对象和 childResources 对象数组，UniversalController 就会对这些资源进行监视，订阅它们的相关事件。
- childResources 对象数组中的每个 childResource 对象都有一个 updateStrategy 对象，通过设置它实现声明式的更新策略，支持 OnDelete, Recreate, InPlace, RollingRecreate, RollingInPlace。
- 调谐接口是唯一需要用户编码实现的模块，用户在这里只需要关心实际业务逻辑，开发完成后将其部署成一个 web 服务，将服务地址填入 UC CRD 的相应字段即可。

3.2.3 动态类型资源处理

Kubernetes 的官方 Go 语言客户端库 client-go，提供了 dynamic 模块，可以用于创建动态客户端，借助于动态客户端，只要知道资源的 apiVersion 和 Kind，就可以对任意一种资源进行操作。UniversalController 基于此，实现了支持动态资源类型的 informer 和 indexer，用于订阅特定资源相关事件以及查找特定类型的资源，是自定义控制器的核心组件。

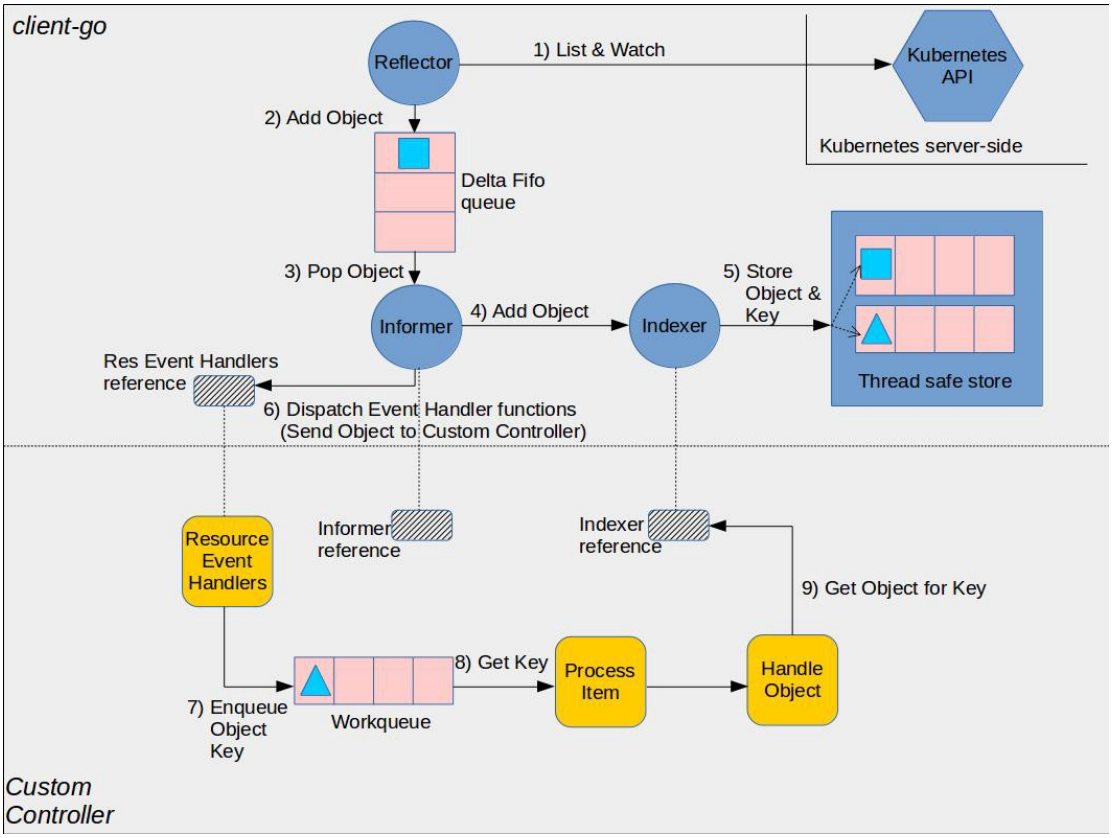

图 3-3: controller 工作原理[2]

3.2.4 控制器设计

图 3-2中的 UC Controller 和 Parent Resource Controller 都是自定义控制器。图 3-3展现了一个自定义控制器的工作方式。

首先介绍一下 client-go 提供给开发者的组件：

- **反射器（Reflector）**：反射器监视着 Kubernetes API 中的指定资源类型（Kind）。完成这项工作的方法（function）是“ListAndWatch”。监视的对象可以是内置的原生资源，也可以是自定义资源。当反射器通过监视 API 收到有新资源诞生的通知时，它使用相应的 listing API 获得该对象，调用“watchHandler”方法，将其放入“Delta FIFO”队列中。
- **通知器（Informer）**：通知器从“Delta FIFO”队列中弹出对象。它的工作是保存对象以便以后检索，并向自定义控制器传递对象。
- **索引器（Indexer）**：索引器提供索引对象的功能。一个典型的索引用例是基于对象标签创建索引。索引器可以基于几个索引功能来维护索引。索引器使用一个线程安全的数据存储来存储对象和它们的键。默认会使用 Store

类型中一个名为 MetaNamespaceKeyFunc 的方法来生成键，该方法将一个对象的键生成为该对象的 < 命名空间 >/< 名称 > 组合。

而自定义控制器中有一下组件：

- **资源事件处理器（Resource Event Handlers）**：资源事件处理器是回调函数，当通知器（Informer）向控制器传递一个对象时，它将被调用。编写这些函数的典型模式是获取被传递对象的键，并将该键排入工作队列（workqueue）等待进一步的处理。
- **工作队列（Workqueue）**：工作队列将对象的传递与处理脱钩，接受到对象后不会立即处理而是放入队列中。
- **处理程序（Process Item）**：处理程序被用于处理工作队列中的项目，它通常使用索引器来检索与键对应的对象。

UC Controller 和 Parent Resource Controller 都使用这种经典控制器模式。

UC Controller 监视着 UC 类型的资源，提供了 UC 相关的服务，同时管理着通过 UC CRD 创建的自定义控制器。它的“Handle Object”部分确保集群状态与 UC CRD 期望的一致，也就是保持所有注册的自定义控制器正确运行。

而 Parent Resource Controller 监视着 Parent Resource，它的“Handle Object 部分”是 UniversalController 用户自定义的代码段，一般是若干个函数。

当开发者使用 UC 的声明式 API 创建控制器时，开发者需要提供的函数中只包含当前控制器所特有的业务逻辑。这些函数会通过 webhook 调用，所以开发者可以用任何能够处理网络请求和 JSON 的编程语言来编写这些函数。

Parent Resource Controller 会执行一个调谐循环，在调谐时调用开发者提供的函数，之后再决定做什么。UniversalController 为每一个 Parent Resource Controller 预先准备了调谐循环的通用逻辑，开发者不需要借助代码生成器，可以完全将精力集中在编写调谐函数上。现阶段 UniversalController 接受的调谐器是 Webhook 形式的，开发者可以借助 serverless 工具，例如 kubeless 或者 openFaas，将函数发布成一个 Web 服务，再提供给控制器。借助 UC 的 API 和 serverless 就可以使开发工作完全集中于业务逻辑，免去了很多琐碎的工作和模版代码。

3.2.5 声明式的调谐器

图 3-2中的“Lambda Hook”模块就是 UniversalController 中的调谐器，以 webhook 的形式实现。

调谐器的工作本质上其实就是两个翻译过程：

- 根据集群的当前状态（state）更新资源的状态（status）。
- 根据资源的 specification 对 Kubernetes 集群进行操作，一般是编排一些 Kubernetes 原生资源，例如 Pods、Services 等，来完成某个应用（例如数据库）的部署与维护。

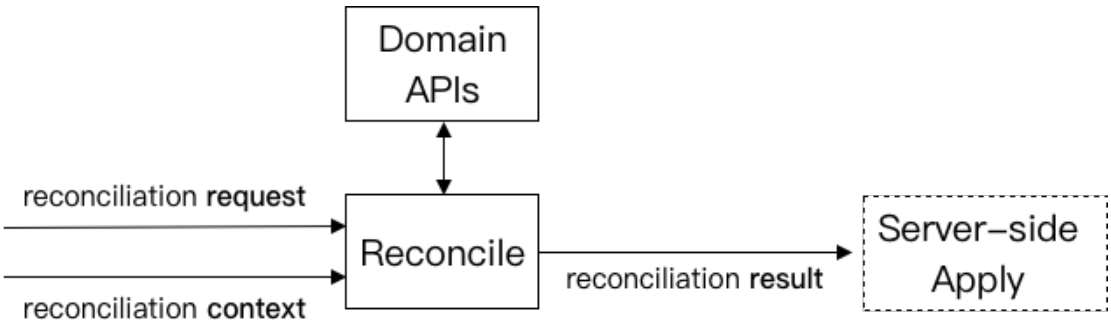

图 3-4: 调谐器

图 3-4是一般的调谐器抽象，它接受调谐请求和当前系统上下文，并返回调谐的结果。在 Kubernetes 中 Domain APIs 就是各种资源，包括 Pod、Service、ReplicaSet 等，用户自定义的资源也包含在其中。在 UC 中，调谐请求就是当前被处理的资源对象，上下文就是通过标签筛选出的子资源以及相关资源，调谐结果是根据当前状态（state）得到该资源对象的状态（status）以及根据该资源对象的规格（specification）得到的期望子资源。

基于 UniversalController 编写的调谐器不用去处理资源的增删改查，不用去与 Kubernetes 交互，只需要去返回期望状态（期望存在的资源集合）。UC Controller 会去决定怎么到达期望状态。这个过程与用户直接通过 kubectl 将资源描述文件提交给 api-server 很接近，指令式的操作极少，也是声明式的。

UniversalController 使用自己的服务端应用（server-side apply）逻辑，它与“kubectl apply”的逻辑接近，遵循惯例而不是配置。开发者不需要在 CRD 中提供如何合并旧资源与新资源的提示，它们的自定义资源和原生资源都根据一套“apply”逻辑得到合并结果。

现阶段调谐器需要以 Webhook 的形式提供，任何可以编写网络服务并处

理 json 的编程语言都可以用于编写这段核心业务逻辑。这是实现语言无关特性的关键。借助 serverless 工具，开发者实际需要编写的只是一个函数而已，其 lambda 表达式为 (parent, children, related) => {... return (status, children)}。

Listing 3.1: sample-controller 中对资源进行增删改查的代码段

```
1 foo, err := c.foosLister.Foos(namespace).Get(name)
2 deployment, err := c.deploymentsLister.Deployments(foo.Namespace)
     .Get(deploymentName)
3 if errors.IsNotFound(err) {
4    deployment, err = c.kubeclientset.AppsV1().Deployments(foo.
        Namespace).Create(context.TODO(), newDeployment(foo),
        metav1.CreateOptions{})
5 }
6 if foo.Spec.Replicas != nil && *foo.Spec.Replicas != *deployment.
     Spec.Replicas {
7    deployment, err = c.kubeclientset.AppsV1().Deployments(foo.
        Namespace).Update(context.TODO(), newDeployment(foo),
        metav1.UpdateOptions{})
8 }
9 _, err := c.sampleclientset.SamplecontrollerV1alpha1().Foos(foo.
     Namespace).Update(context.TODO(), fooCopy, metav1.
     UpdateOptions{})
```

3.2.6 声明式的更新策略

Kubernetes 的原生资源 Deployment 和 StatefulSet 都有对它们管理的 Pod 进行滚动更新的功能。UniversalController 也支持滚动更新，内置了多个可选的更新策略，并且可以通过声明式使用，开发者只需要在 UC 资源中填入相应字段就可以让自己的控制器拥有滚动更新的能力。

3.3 小结

本章首先对一个通用 Opeator 需要实现的功能进行了介绍，然后在需求分析的基础之上对 UniversalController 的总体架构和各个模块进行了详细说明。

第四章　声明式的通用 Kubernetes Operator 的实现

UnversalController 自身依然是一个传统的 Operator，用 Go 语言编写完成。因为 UniversalController 不能事先确定自己需要监视和处理的资源类型，所以使用了 Kubernetes 官方 Go 语言客户端 client-go 的 dynamic 包，它提供了动态的客户端，可以操作任意类型的资源，包括原生资源以及用户自定义资源，这是它作为一个通用 Operator 的关键。

4.1　自定义资源

Listing 4.1: UC CRD

```
1  apiVersion: apiextensions.k8s.io/v1
2  kind: CustomResourceDefinition
3  metadata:
4    annotations:
5      "api-approved.kubernetes.io": "unapproved, request not yet
           submitted"
6    name: "universalcontrollers.universalcontroller.njuics.cn"
7  spec:
8    group: "universalcontroller.njuics.cn"
9    names:
10     kind: UniversalController
11     listKind: UniversalControllerList
12     plural: universalcontrollers
13     shortNames:
14     - uc
15     - uctl
16     singular: universalcontroller
17   scope: Cluster
```

```
18 ...
```

YAML 代码 4.1是 UC CRD 的定义，通过 kubectl 向 api-server 提交这个 yaml 文件就可以在 Kubernetes 中注册一种新的资源类型“UniversalController”。这样就成功拓展了 Kubernetes APIs，之后就可以使用这个新的 API 提交资源定义来注册控制器。

Listing 4.2: 作为 UC CRD 示例的 catset-controlelr

```
1  apiVersion: universalcontroller .njuics .cn/v1alpha1
2  kind: UniversalController
3  metadata:
4    name: catset- controller
5  spec:
6    parentResource:
7      apiVersion: mlhub.njuics .cn/v1alpha1
8      resource: catsets
9    childResources:
10     - apiVersion: v1
11       resource: pods
12       updateStrategy:
13         method: RollingRecreate
14         statusChecks:
15           conditions:
16             - type: Ready
17               status: "True"
18     - apiVersion: v1
19       resource: persistentvolumeclaims
20   hooks:
21     sync:
22       webhook:
23         url: "http://catset-ctrl.universalcontroller:8080"
```

YAML 文件 4.2是一个 UC CRD 资源的例子，定义了 CatSet 资源的控制器，它的行为几乎与 Kubernetes 原生资源的 StatefulSet 的控制器一致，是对 StatefulSet 的二次实现。

一个 UC CRD 的“spec”有如下字段：

- parentResource：类型是 ResourceRule，用于指定父资源类型，也就是这个

控制器实际管理的资源。ResourceRule 只有两个字段“APIVersion”和“Resource”，用于确定一种资源类型。

- childResources：是一组 ResourceRule，用于指定会被控制器生成的子资源类型。
- resyncPeriodSeconds：规定两次调谐之间间隔的时间，设置后就算工作队列当前是空的，但是距离上次调谐过去这么多时间后还是会去调谐。
- generateSelector：bool 型，如果为真，那么忽略父资源对象的标签选择器（如果存在的话），UniversalController 会为其生成独一无二的标签选择器，避免与其他的资源冲突。这里的标签选择器都是为了筛选子资源，符合标签选择器的子资源都会被当做父资源的子资源。
- hooks：一组定义了控制器行为的 lambda hook。

4.2 动态类型高级操作接口实现

Listing 4.3: 客户端实现

```
type Clientset struct {
    config  rest.Config
    resources *dynamicdiscovery.ResourceMap
    dc      dynamic.Interface
}
type ResourceClient struct {
    dynamic.ResourceInterface
    *dynamicdiscovery.APIResource
    rootClient dynamic.NamespaceableResourceInterface
}
func New(config *rest.Config, resources *dynamicdiscovery.
    ResourceMap) (*Clientset, error)
func (cs *Clientset) Resource(apiVersion, resource string) (*
    ResourceClient, error) {
    // Look up the requested resource in discovery.
    apiResource := cs.resources.Get(apiVersion, resource)
    if apiResource == nil {
        return nil, fmt.Errorf("discovery: can't find resource %s
            in apiVersion %s", resource, apiVersion)
```

```
17      }
18      return cs.resource(apiResource), nil
19  }
20  func (cs *Clientset) resource(apiResource *dynamicdiscovery.
        APIResource) *ResourceClient {
21      client := cs.dc.Resource(apiResource.GroupVersionResource())
22      return &ResourceClient{
23          ResourceInterface: client,
24          APIResource:  apiResource,
25          rootClient:   client,
26      }
27  }
28  func (rc *ResourceClient) AtomicUpdate(orig *unstructured.
        Unstructured, update func(obj *unstructured.Unstructured) bool
        ) (result *unstructured.Unstructured, err error)
29
30  type Unstructured struct {
31      // Object is a JSON compatible map with string, float, int,
            bool, []interface{}, or
32      // map[string]interface{}
33      // children.
34      Object map[string]interface{}
35  }
```

代码段4.3是客户端的结构体和一些方法，UC 首先需要用 New 方法得到一个 Clientset 类型的对象，对于各种资源，只要知道资源的 apiVersion 和 kind，都可以用这个对象的 Resource 方法得到一个 ResourceClient 对象，该对象可以对这类资源进行各类 CURD 的操作，所有的资源都以 Unstructured 类型存储，Unstructured 中只有一个字典类型的字段，实际是把资源以接近 JSON 的形式存储起来。

Listing 4.4: 通知器（Informer）实现

```
1  // sharedResourceInformer is the actual, single informer that's
       shared by
2  // multiple ResourceInformer instances.
3  type sharedResourceInformer struct {
4      informer cache.SharedIndexInformer
```

```
5     lister dynamiclister.Lister
6     defaultResyncPeriod time.Duration
7     eventHandlers *sharedEventHandler
8     close func()
9  }
10 func newSharedResourceInformer(client *dynamicclientset.
      ResourceClient, defaultResyncPeriod time.Duration, close func
      ()) *sharedResourceInformer {
11    informer := cache.NewSharedIndexInformer(
12       &cache.ListWatch{
13          ListFunc: func(opts metav1.ListOptions) (runtime.Object,
                error) {
14             return client.List(opts)
15          },
16          WatchFunc: client.Watch,
17       },
18       &unstructured.Unstructured{},
19       defaultResyncPeriod,
20       cache.Indexers{
21          cache.NamespaceIndex: cache.MetaNamespaceIndexFunc,
22       },
23    )
24    sri := &sharedResourceInformer{
25       close:           close,
26       informer:        informer,
27       defaultResyncPeriod: defaultResyncPeriod,
28       lister: dynamiclister.New(informer.GetIndexer(), client.
            GroupVersionResource()),
29    }
30    sri.eventHandlers = newSharedEventHandler(sri.lister,
         defaultResyncPeriod)
31    informer.AddEventHandler(sri.eventHandlers)
32    return sri
33 }
34 type ResourceInformer struct {
35    sharedResourceInformer *sharedResourceInformer
36    informerWrapper   *informerWrapper
```

```
37 }
38 func newResourceInformer(sri *sharedResourceInformer) *
      ResourceInformer {
39    return &ResourceInformer{
40       sharedResourceInformer: sri,
41       informerWrapper: &informerWrapper{
42          SharedIndexInformer: sri.informer,
43          sharedResourceInformer: sri,
44       },
45    }
46 }
```

代码段4.4展示了如何在 client 的基础之上实现 informer，informer 会调用 client 的 List 和 Watch 方法来监听资源。SharedInformer 的作用是让在一个进程中运行的控制器们共享订阅，避免重复订阅浪费内存和网络带宽。

4.3 控制器实现

一个 UC CRD 被提交后就相当于在 UniversalController 中注册了一个新的控制器。

UC Controller 和 Parent Resource Controller 的资源事件处理器（Resource Event Handlers）、工作队列（Workqueue）、处理程序（Process Item）都符合经典的控制器模式。在3.2.4小节已经详细介绍，这里不再赘述。

接下来介绍 UC Controller 和 Parent Resource Controller 的“Handle Object”分别是怎么实现的。

4.3.1 同步 UC CRD 资源

Listing 4.5: 同步 UC CRD

```
1 func (u *Universalcontroller) syncUniversalController(uc *
     v1alpha1.
2    UniversalController) error {
3    if pc, ok := u.parentControllers[uc.Name]; ok {
4       if apiequality.Semantic.DeepEqual(uc.Spec, pc.uc.Spec) {
5          // Nothing has changed.
```

```
6            return nil
7        }
8        pc.Stop()
9        delete(u.parentControllers, uc.Name)
10   }
11   pc, err := newParentController(u.resources, u.dynClient, u.
         dynInformers,
12       u.mcClient, u.revisionLister, uc, u.numWorkers)
13   if err != nil {
14       return err
15   }
16   pc.Start()
17   u.parentControllers[uc.Name] = pc
18   return nil
19 }
```

代码段4.5中的 syncUniversalController 方法根据当前的 UC CRD 资源定义进行调谐，如果这个资源对应的控制器已经存在，并且不需要修改，spec 完全一致，那么不用做任何事，本次调谐结束，否则就删除旧的控制器。接下来新建 Parent Resource 的控制器，并启动，和一般的 controller-mananger 模式中一样，这个 Parent Resource 的控制器运行在单独的一个 Go 协程中，只是此时 UC Controller 承担了 manager 的职责，所以 UC Controller 是 controller-controller。

4.3.2 同步父资源（Parent Resource）

代码段4.6展示了 Parent Resource Controller 对 Parent Resource 的同步过程：

1. claimChildren 方法会通过标签选择器找到所有的已经存在的子资源。
2. 如果 Customize Hook 非空，通过 Customize Hook 找到相关资源。
3. 将 Parent Resource、Child Resources、Related Resources 放入同步钩子请求体中，调用同步钩子，得到同步结果，其中包含期望的 Parent Resource Status 和期望的 Child Resources，
4. 先比较已经存在的 Child Resources 和期望的 Child Resources，如果一个资源已经存在，但是与期望不一致，就把它们两个当做 JSON 对象合并，之后根据更新策略更新得到合并结果；如果一个期望的资源还不存在，就创建它，其实就是执行了“Server-side apply”。

5. 最后更新 Parent Resource Status。

Listing 4.6: 同步父资源

```
1  func (pc *parentController) syncParentObject(parent *unstructured
       .Unstructured) error {
2      observedChildren, err := pc.claimChildren(parent)
3      if err != nil {
4          return err
5      }
6      relatedObjects, err := pc.customize.GetRelatedObjects(parent)
7      if err != nil {
8          return err
9      }
10     syncResult, err := pc.syncRevisions(parent, observedChildren,
           relatedObjects)
11     if err != nil {
12         return err
13     }
14     desiredChildren := common.MakeChildMap(parent, syncResult.
           Children)
15     if syncResult.ResyncAfterSeconds > 0 {
16         pc.enqueueParentObjectAfter(parent, time.Duration(
               syncResult.ResyncAfterSeconds*float64(time.Second)))
17     }
18     var manageErr error
19     if parent.GetDeletionTimestamp() == nil || pc.finalizer.
           ShouldFinalize(parent) {
20         // Reconcile children.
21         if err := common.ManageChildren(pc.dynClient, pc.
               updateStrategy, parent, observedChildren,
22             desiredChildren); err != nil {
23             manageErr = fmt.Errorf("can't reconcile children for %v
                   %v/%v: %v",
24                 pc.parentResource.Kind, parent.GetNamespace(), parent
                       .GetName(), err)
25         }
26     }
```

```
27      if _, err := pc.updateParentStatus(parent, syncResult.Status);
            err != nil {
28          return fmt.Errorf("can't update status for %v %v/%v: %v",
                pc.parentResource.Kind,
29              parent.GetNamespace(), parent.GetName(), err)
30      }
31      return manageErr
32  }
```

4.4 声明式的更新策略

4.4.1 更新策略介绍

UniversalController 提供了很多更新策略，开发者可以通过声明式接口使用它们，而不用写任何代码。

- 待删除后更新（OnDelete）：不更新现有的子资源，直到它被其他的客户端例如 kubectl 删除。
- 立刻重建（ReCreate）：立即删除任何不符合期望状态（state）不同的子资源，并根据状态（state）下重新创建。
- 就地更新（InPlace）：立刻就地更新任何不符合期望状态（state）不同的子资源。
- 滚动重建（RollingRecreate）：每次调谐删除一个与期望状态（state）不同的子资源，并在处理下一个子资源之前根据期望状态（state）重建它。在任意时刻，如果已经更新的子资源中有一个或多个状态检查失败，则暂停滚动更新。
- 滚动就地更新（RollingInPlace）：每次更新一个与期望状态（state）不同的子资源。如果已经更新的子资源中有一个或多个状态检查失败，则暂停滚动更新。

不同的资源适合不同的更新策略，例如 Pod 一般会用 ReCreate 或者 RollingRecreate，因为当一个已经在 Kubernetes 中存在的 Pod，它能修改的只有 metadata 中的标签（labels）和附加说明（annotations），spec 中的字段都不能修改，如果需要修改，就只能删除重建。而 Deployment 除了名字和命名空间都可以修改，用 InPlace 或者 RollingInPlace 显然更合适。

4.4.2 滚动更新版本控制

ControllerRevision 是 UniversalController 使用的一个内部 API，用于实现声明式的滚动更新，主要受到 Kubernetes 原生资源 StatefulSet 和 DaemonSet 使用的 ControllerRevision 启发后实现。

每个 ControllerRevision 都与一个资源相关，名称是该资源的类型、资源所在的 apiGroup 以及版本后缀组成的。版本后缀是对该资源指定字段的哈希结果。

默认情况下，一旦一个特定的父资源被删除，属于该资源的 ControllerRevision 们会被垃圾回收处理掉。但是，也可以在父资源的删除过程中抛弃 ControllerRevision，不再与这个父资源有关系的 ControllerRevision 也就不会被删除了，就可以创建另一个父资源来接管它。接管的规则基于父资源的标签选择器，和 ReplicaSet 接管 Pods 的方式一样。

Listing 4.7: ControllerRevision 示例

```
1  ---
2  apiVersion: universalcontroller.njuics.cn/v1alpha1
3  kind: ControllerRevision
4  metadata:
5    name: catsets.universalcontroller.njuics.cn-5463
         ba99b804a121d35d14a5ab74546d1e8ba953
6    labels:
7      app: nginx
8      component: backend
9      universalcontroller.njuics.cn/apiGroup: universalcontroller.njuics.cn
10     universalcontroller.njuics.cn/resource: catsets
11 parentPatch:
12   spec:
13     template:
14       [...]
15 children:
16 - apiGroup: ""
17   kind: Pod
18   names:
19   - nginx-backend-0
20   - nginx-backend-1
```

```
21   - nginx-backend-2
```

YAML 文件 4.7是 ControllerRevision 的一个例子，parentPatch 字段存储了父资源的部分表示，它只包含 UC CRD 的 revisionHistory 字段列出的那些参与滚动更新的字段，默认是 spec。

例如，如果一个 UC CRD 的 revisionHistory 是数组 ["spec.template"]，那么 parentPath 只会包含 spec.template 和嵌套在其中的子字段。

这样就可以在滚动更新的过程中做出选择性行为。任何不属于 revisionHistory 的字段如果被更新，更新都会立即生效，而不是进行滚动更新。

children 字段存储了一个"属于"这个 ControllerRevison 的子资源列表，UniversalController 就是通过这个字段跟踪一个子资源属于哪个 ControllerRevision。在滚动更新过程中，如果一个还没有更新的 Pod 被用户通过 kubectl 删除了，那么它应该重建它被删除之前的版本，而不是最新版本，以保证滚动更新的粒度与次序不被打乱。

当 UniversalController 决定将一个子资源更新到另一个版本时。它首先会更新相关的 ControllerRevison 来表达这个意图，这些更新被提交后，它就会根据所配置的子资源更新策略开始更新该子资源。这确保了滚动更新的中间结果在 api-server 中被持久化，就算 UniversalController 重启，也能从中断的位置继续更新。

children 字段的值是按照 apiGroup 和 kind 进行分组的。对于每个 apiGroup 和 kind 的组合，存储了一个对象名称列表。

4.5 调谐器接口实现

在示例的 YAML 文件 4.2中 hooks.sync 就定义了当前控制器使用的调谐器。编写调谐器是在开发者基于 UniversalController 开发 Operator 时唯一需要的自定义代码编写工作。编写完成后需要以 webhook 的形式发布出来，借助 serverless 工具即可省去网络相关代码的编写。之后再 UC CRD 的相应字段填写服务地址就完成了控制器的配置。UC CRD 中的 Webhook 结构如表 4-1所示。在 webhook 中，service 的结构如表 4-2所示。

Hooks

在 UC CRD 的 spec 中，hooks 字段有以下三个子字段。

表 4-1: Webhook

字段	Go 类型	说明
url	string	完整的 url 地址，优先级比 path 和 service 的组合高
timeout	Duration	时限，过期未收到回复就是请求超时
path	string	请求链接的后缀
service	ServiceReference	应该被发送请求的 K8s Service

表 4-2: Service Reference

字段	Go 类型	说明
name	string	该 Service 的名称
namespace	string	该 Service 的命名空间
port	int32	该 Service 提供服务的接口
protocol	string	协议，默认为 http

- sync：用于指定如何调用同步钩子。
- finalize：用于指定如何调用收尾（finalize）钩子。
- customize：用于指定如何调用 Customize 钩子

这每个字段都对应了一种 Hook 类型，下面开始分别介绍。

同步钩子

同步钩子被用来指定为给定的父资源创建或维护那些子资源，即期望状态（state）。根据 UC CRD 的 spec，UniversalController 会收集所有需要的资源，并向同步钩子发送最新观察到的状态（state）。同步钩子返回期望状态后，UniversalController 会开始通过一系列操作向它收敛，操作包括适当地创建、删除和更新对象。

可以简单的把同步钩子看做一个脚本，它生成 json 发送到“kubectl apply”，同时，与一次性的客户端生成器不同的是，这个脚本可以观察到集群中最新的状态（state），并且会在观察到的状态（state）发生变化时自动被执行。

同步钩子请求

一个请求中只会包含一个父资源，所以同步钩子一次只需要考虑一个父资源。

请求体是一个 JSON 对象，它有以下字段：

- **parent**：parent 对象是一个 json 形式的父资源，和用 kubectl get <parent-resource> <parent-name> -o json 得到的结果一样。
- **children**：children 对象存储了与父资源相关的子资源们，是通过标签选择器筛选得到的。
- **related**：只有当 Customize 钩子存在时，related 对象会存储相关资源，否则为空。
- **finalizing**：布尔值，在调用同步钩子是始终为 false。

children 对象的每个字段都代表 UC CRD 的 spec 中指定的子资源类型之一。每个子资源类型的字段名是 <kind>.<apiVersion>。举例来说，Pods 的字段名是 Pod.v1，而 StatefulSets 的字段名是 StatefulSet.apps/v1。

在每个字段中（例如在 children['Pod.v1'] 中），存储这一个字典，键是当前资源标识，值是该资源的 json 表示。如果父资源和子资源的作用域相同，都是集群的或者都是命名空间的，那么键就只是子资源的名称，如果父资源是集群作用域，而子资源是命名空间作用域，那么键的形式是.metadata.namespace/.metadata.name。这是为了区分可能存在的在不同命名空间的两个同名子资源。父资源是命名空间作用域而子资源是集群作用域的情况不可能出现。举例来说，如果父资源在 my-namespace 命名空间下，那么在 my-namespace 命名空间下的一个名称为 my-pod 的 Pod 会被存储在 request.children['Pod.v1']['my-pod']。如果父资源是集群作用域的，这个 Pod 会被存储在 request.children['Pod.v1']['my-namespace/my-pod']。每个子资源类型总是有一个入口，即使在同步时没有观察到该类型的子资源。例如，如果 Pod 是子资源类型之一，但没有任何现有的 Pods 资源与父资源的选择器相匹配，请求体的形式是：

Listing 4.8: 请求体

```
{
  "children": {
    "Pod.v1": {}
  }
}
```

而不是

Listing 4.9: 异常请求体

```
{
  "children": {}
}
```

related 字段下存储着相关资源对象，格式与 children 字段下的对象相同，表示与给定父资源的 Customize 钩子响应相匹配的资源，这些资源不由控制器管理，因此不可修改，但可以将它们当做系统上下文，进而得到子资源的期望配置。当观察到相关资源被更新时，就算父资源和子资源都没有变化，同步钩子也会被触发。

同步钩子响应

同步钩子的响应体有如下字段：

- **status**：一个 JSON 对象，将完全取代父资源中的状态字段。
- **children**：一个 JSON 对象的列表，代表所有期望存在的子资源。
- **resyncAfterSeconds**：下次同步的时间间隔，以秒为单位，类型是浮点数。

状态（status）的设置完全由用户代码段决定，状态（status）应该根据最后观察到的状态（state）来填写，是一个当前值，而不是期望值。

响应体中的 children 字段是一个对象数组，而不是请求体中那样的字典，每一个对象都是一个期望存在的子资源。UniversalController 按照类型和名称对发送的对象进行分组，以方便用户简化脚本，但实际上这是多余的，因为每个对象都包含自己的 apiVersion、kind 和 metadata.name。

任何在请求体中存在的子资源，如果用户代码拒绝在响应体中返回，它会被 UniversalController 在收到响应后删除。但是，用户不应该直接把请求中的子资源复制到返回结果中，因为它们的形式不同，返回的结果应该完全根据父资源的 specification 和系统上下文重新生成。用户应该把响应体中的每个子资源看作是被发送到“kubectl apply”，只需要设置用户关心的字段。

如果返回的 resyncAfterSeconds 被设置为一个大于 0 的值，同步钩子在延迟一段时间后会被再次调用，请求体中的 parent 字段的值依然是这个特定的父资源，其他字段依据父资源设置。这个设置时一次性的，不会周期性重新同步，而且只针对这个特定的父资源。

Finalize 钩子

如果定义了 finalize 钩子，UniversalController 将为父资源添加一个 finalizer，这将防止它被直接删除，直到 finilize 钩子执行完，并且钩子的响应表明清理已经完成，它才能真正的被删除。

这对于清理可能在外部系统中创建的资源是很有用的。如果定义 finalize 钩子，那么当一个父对象被删除时，垃圾回收器会立即删除所有的子对象，而不会调用任何钩子。

finalize 钩子的语义大多与同步钩子的语义相当。UniversalController 将尝试调谐在 children 字段中返回的期望状态（state），并将在父资源上设置状态（status）。主要的区别是，当父资源正在被删除且需要清理时，会调用 finalize 钩子而不是同步钩子。

当观察到的状态发生变化时，UniversalController 可能会多次调用 finalize 钩子，甚至可能是在一次表明已经完成 finalize 的调用之后。用户编写的处理程序应该知道如何检查还需要做什么，如果没有什么需要做的，就报告成功。

同步钩子和 finalize 钩子都有一个叫做 finalizing 的请求字段，用来指示到底调用了哪个钩子，在同步钩子请求中始终为 false，在 finalize 钩子请求中始终为 true。这让用户可以自己选择将 finalize 钩子作为一个单独的处理程序还是作为同步处理程序中的一个分支来实现，这取决于它们共享多少逻辑。要为两者使用相同的处理程序，只需定义一个 finalize 钩子，并将其设置为与同步钩子相同的值。

finalize 钩子请求

finalize 钩子的请求体格式与同步钩子的完全相同，只是 finalizing 字段始终为 true 而已。如果同步钩子和 finalize 钩子共享同一段处理程序，可以使用 finalizing 字段来判断是该清理还是进行正常的同步。如果为 finalize 定义了一个单独的处理程序，就不需要检查 finalizing 字段，因为它总是为真。

finalize 钩子响应

finalize 钩子响应体拥有所有同步钩子响应体的字段，但还有一个额外的字段 finalized，是一个布尔值，用于表示清理是否已经结束。

Customize 钩子

如果定义了 Customize 钩子，UniversalController 会询问它哪些资源是相关资源，应该放入同步钩子和 finalize 钩子的请求中。这在有些场景下非常有

用。一个例子是，用户想实现一个控制器将指定的 ConfigMaps 复制到每个 Namespace 中。另一个例子是有些控制器希望能够引用相关对象的一些信息，例如，从有些 Pod 资源获取 env 部分。如果没有定义 Customize 钩子，那么同步钩子和 finalize 钩子的请求体中 related 都将是空的。

当前 Customize 钩子的请求体中不会提供任何关于集群当前状态（state）的信息，只包含父资源，所以相关对象的集合只取决于父资源的 spec。

Customize 钩子请求

Customize 钩子的请求体只有一个 parent 字段，用于存储一个父资源的 json 表示。

Customize 钩子响应

Customize 钩子的响应体只有一个字段“relatedResources”，存放了一组 JSON 对象，每个 JSON 对象是一个 ResourceRule，用于筛选资源。

ResourceRule 有以下字段：

- **apiVersion**：资源的 apiVersion，例如 apps/v1、v1、batch/v1 等。
- **resource**：资源的小写名称，例如 deployments, replicasets, statefulsets。
- **labelSelector**：用于筛选资源的标签选择器，如果为空，用 namespace 和 names 字段去定位资源。
- **namespace**：选填项，资源所在的命名空间。
- **name**：选填项，资源名称列表。

如果设置了 labelSelector，namspace 字段和 name 字段就应该都为空，反之亦然，它们不应该被同时设置，它们代表了两种不同的资源筛选方式，在一次筛选中只能使用一种。

UniversalController 收到 Customize 钩子响应后就回去用这一系列资源筛选规则找到符合调节的资源们，并放入同步或 finalize 钩子的请求体中。

4.6 小结

本章详细介绍了各个组件或功能在 UniversalController 中是如何用 Go 语言实现的，涉及到了各方面的细节。

第五章　实验评估

5.1　用例 1：重新实现 sample-controller

5.1.1　介绍

sample-controller 是 Kubernetes 官方提供的一个 Operator 编写样例，项目地址是 https://github.com/kubernetes/sample-controller。

它是一个简单的控制器，监视通过 CustomResourceDefinition 定义的 Foo 资源，为每个 Foo 保证一个对应的 Deployment 存在。sample-controller 展示了一个标准的 Operator 是如何实现并工作的，使用 client-go 与 Kubernetes api-server 交互，编写控制器的各个组件，没有使用更高级的抽象包。

5.1.2　实现

Listing 5.1: sample-controller 的实现代码

```
module.exports = {
 handler: (event, context) => {
   let observed = event['data'];
   let desired = {status: {}, children: []};
   let foo = observed.parent;// observed foo object
   // extract available replicas from desired deployment if
      available
   let allDeploys = observed.children['Deployment.apps/v1'];
   let fooDeploy = allDeploys ? allDeploys[foo.spec.
      deploymentName] : null;
   let replicas = fooDeploy ? fooDeploy.status.availableReplicas
      : 0;
   desired.status = {availableReplicas: replicas};// Set the
      status of Foo
   desired.children = [desiredDeployment(foo)];
```

```
    return desired;
  }
};
var desiredDeployment = function (foo) {
  let lbls = {app: "sample"};
  let deploy = {
    apiVersion: "apps/v1",
    kind: "Deployment",
    metadata: {
      name: foo.spec.deploymentName,
      namespace: foo.metadata.namespace,
    },
    spec: {
      replicas: foo.spec.replicas,
      selector: {matchLabels: lbls},
      template: {
        metadata: {labels: lbls},
        spec: {
          containers: [{
              name: "nginx",
              image: "nginx:stable"
            }]
        }
      }
    }
  };
  return deploy;
};
```

JavaScript 代码5.1就是所有需要写的代码，而不是一个代码段。它的逻辑很简单，从请求体中取出 Foo 资源，根据它的定义生成期望的 Deployment，Foo 资源的 status 只有一个字段表示可用的副本数，如果 Deployment 还不存在，可用副本数为 0，否则就是该 Deployment 的 status.availableReplicas 的值，最后将 status 和 Deployment 返回即可。

接下来借助 kubeless 将这个函数部署成 web 服务，只需要一条命令：kubeless -n universalcontroller function deploy sample-controller –runtime nodejs10

–from-file sync.js –handler sync.handler

之后在 universalcontroller 命名空间下会生成一个名叫 sample-controller 的 Service 资源和一个名叫 sample-controller 的 Deployment 资源，于是在集群内部就可以用 http://sample-controller.universalcontroller:8080 访问这个服务，这个 url 就是要生成的 controller 使用的同步钩子。

Listing 5.2: sample-controller 的配置文件

```
apiVersion: universalcontroller.njuics.cn/v1alpha1
kind: UniversalController
metadata:
  name: sample-controller
spec:
  generateSelector: true
  parentResource:
    apiVersion: njuics.cn/v1alpha1
    resource: foos
  childResources:
  - apiVersion: apps/v1
    resource: deployments
    updateStrategy:
      method: InPlace
  hooks:
    sync:
      webhook:
        url: "http://sample-controller.universalcontroller:8080"
```

Listing 5.3: sample-controller 的配置文件

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  volumes:
    - name: example-storage
      persistentVolumeClaim:
        claimName: example-claim
  containers:
    - name: example-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: example-storage
```

Listing 5.4: sample-controller 的配置文件

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
```

最后用 kubectl 提交 YAML 文件 5.4，就可以在 UniversalController 中注册一个控制器

5.1.3 对比总结

原来的 sample-controller 是用 Go 语言实现的，总代码行数为 1701，剔除使用代码生成工具生成的代码后代码行数为 739。而借助 UniversalController，可以用 JavaScript 来编写业务逻辑，并且 controller 配置和代码加起来也只有 58 行。图 5-1很直观的显示了工作量的巨大差距。对于功能简单的 Operator，借助 UniversalController 可以实现代码量很少的极速开发。

5.2 用例 2：重新实现 tf-operaotr

5.2.1 介绍

tf-operator 由 kubeflow 社区开发，项目地址为 https://github.com/kubeflow/tf-operator，它提供了 TFJob 这个 Kubernetes 自定义资源，使其能够轻松地在 Kubernetes 上运行分布式或非分布式 TensorFlow 任务。

5.2.2 实现

Listing 5.5: tensorflowjob-controller 的实现代码

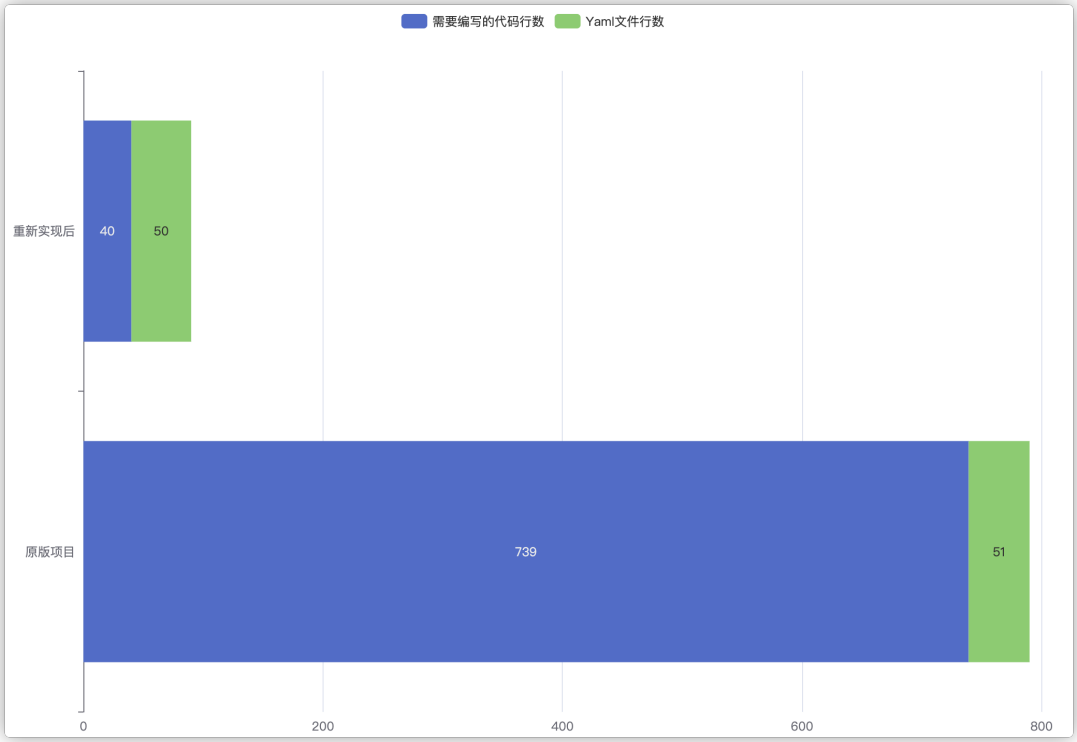

图 5-1: sample-controller 原版与 UC 版代码行数比较

```
let tfjob = observed.parent;
let status = deepCopy(tfjob.status);
let observedPods = Object.values(observed.children['Pod.v1']);
let replicas = tfjob.spec.tfReplicaSpecs;
for (let rtype of Object.keys(replicas)) {
  let spec = replicas[rtype];
  let rt = rtype.toLowerCase();
  status.replicaStatuses[rtype] = {active: 0, succeeded: 0, failed
      : 0};
  let pods = filterPodsForReplicaType(observedPods, rt);
  let numReplicas = spec.replicas;
  let podSlices = getPodSlices(pods, numReplicas);
  for (let index = 0; index < podSlices.length; index++) {
    if (index < numReplicas) {
      desired.children.push(newSVC(tfjob, rtype, index));
    }
    let podSlice = podSlices[index];
```

```
17    if (podSlice.length === 1) {
18      let pod = podSLice[0];
19      let exitCode = getContainerExitCode(pod);
20      if (spec.restartPolicy === 'ExitCode') {
21        if (pod.status.phase === 'Failed' && isRetryableExitCode(
              exitCode)) {
22          updateJobConditions(status, 'Restarting', 'TFJobRestarting
              ',
23              `TFJob ${tfjob.metadata.name} is restarting because ${
                  rtype} replica(s) failed.`);
24          continue;
25        }
26      }
27      if (pod.status.phase === 'Running') {
28        status.replicaStatuses[rtype].active++;
29      } else if (pod.status.phase === 'Succeeded') {
30        status.replicaStatuses[rtype].succeeded++;
31      } else if (pod.status.phase === 'Failed') {
32        status.replicaStatuses[rtype].failed++;
33      }
34    }
35    if (index < numReplicas) {
36      let masterRole = isMasterRole(replicas, rtype, index);
37      desired.children.push(newPod(tfjob, rt, index, spec,
            masterRole, replicas));
38    }
39  }
40  }
```

tf-operator 依然用 JavaScript 实现，但实现逻辑相当复杂，总代码行数为 408 行，核心代码为代码段 5.5。原版 tf-operator 中的逻辑都转译了过来，包括根据不同的副本类型使用当前 tfjob 资源中相应的模版来创建 Pod，为每一个 Pod 创建一个 Service，分别统计每种副本的状态来更新 tfjob 的 status 等。

接下来借助 kubeless 将这个函数部署成 web 服务，部署命令为：

kubeless -n universalcontroller function deploy tensorflow-controller –runtime nodejs10 –from-file sync1.js –handler sync1.handler

之后在 universalcontroller 命名空间下会生成一个名叫 tensorflow-controller 的 Service 资源和一个名叫 tensorflow-controller 的 Deployment 资源，于是在集群内部就可以用 http://tensorflow-controller.universalcontroller:8080 访问这个服务，这个 url 就是要生成的 controller 使用的同步钩子。

Listing 5.6: tensorflowjob-controller 的配置文件

```
1  apiVersion: universalcontroller .njuics .cn/v1alpha1
2  kind: UniversalController
3  metadata:
4    name: tensorflowjob- controller
5  spec:
6    parentResource:
7      apiVersion: mlhub.njuics .cn/v1alpha1
8      resource: tensorflowjobs
9    childResources:
10     - apiVersion: v1
11       resource: services
12       updateStrategy:
13         method: InPlace
14     - apiVersion: v1
15       resource: pods
16       updateStrategy:
17         method: ReCreate
18   hooks:
19     sync:
20       webhook:
21         url: "http://tensorflow-controller.universalcontroller:8080"
```

YAML 文件 5.6是 tensorflowjob-controller 的声明式定义。

5.2.3 对比总结

原版 tf-operator 主要使用了 client-go 包，实现了一个标准的 Operator，Go 代码行数为 17155，剔除使用代码生成工具生成的代码以及测试代码后代码行数为 2344。在 UniversalController 之上实现的版本只需要四百多行 JavaScript 代码就能实现同样的功能。这个用例主要是为了展示 UniversalController 具备在实

现复杂 Operator 时依然保持工作量相对较小的能力。图 5-2直观地展现了工作量的差距。

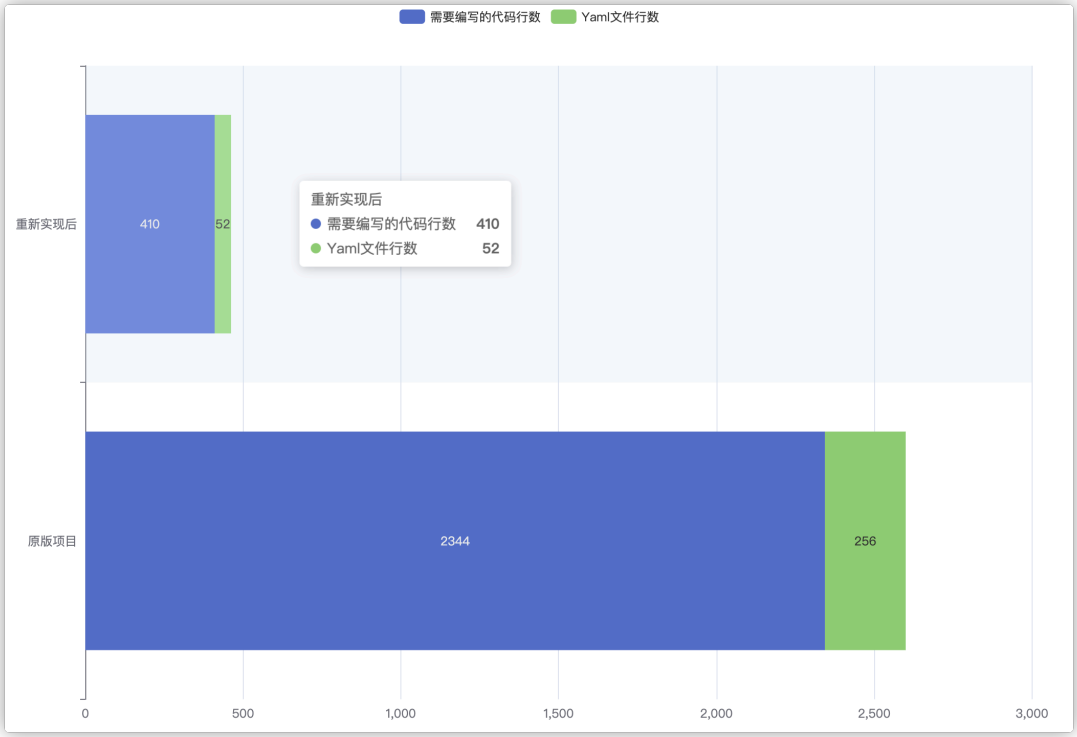

图 5-2: tf-operator 原版与 UC 版代码行数比较

5.3 用例 3：CatSet 与滚动更新

5.3.1 介绍

CatSet 是对 Kubernetes 原生资源 StatefulSet 的重新实现，同时可以展示滚动更新的使用。

5.3.2 实现

Listing 5.7: 生成 Pod 和 PVC

```
module.exports = {
  handler: (event, context) => {
    let observed = event.data;
```

```
  let desired = {status: {}, children: []};
  let catset = observed.parent;
  // Arrange observed Pods by ordinal.
  let observedPods = {};
  if (observed.children && observed.children['Pod.v1']) {
    for (let pod of Object.values(observed.children['Pod.v1'])) {
      let ordinal = getOrdinal(catset.metadata.name, pod.metadata
          .name);
      if (ordinal >= 0) observedPods[ordinal] = pod;
    }
  }
  if (observed.finalizing) {
    // If the parent is being deleted, scale down to zero
        replicas.
    catset.spec.replicas = 0;
    // Mark the finalizer as done if there are no more Pods.
    desired.finalized = (Object.keys(observedPods).length === 0);
  }
  for (var ready = 0; ready < catset.spec.replicas &&
      isRunningAndReady(observedPods[ready]); ready++) ;
  desired.status = {replicas: Object.keys(observedPods).length,
      readyReplicas: ready};
  let desiredPods = {};
  for (let ordinal in observedPods) {
    desiredPods[ordinal] = newPod(catset, ordinal);
  }
  // Fill in one missing Pod if all lower ordinals are Ready.
  if (ready < catset.spec.replicas && !(ready in desiredPods)) {
    desiredPods[ready] = newPod(catset, ready);
  }
  // If all desired Pods are Ready, see if we need to scale down
      .
  if (ready === catset.spec.replicas) {
    let maxOrdinal = Math.max(...Object.keys(desiredPods));
    if (maxOrdinal >= catset.spec.replicas) {
      delete desiredPods[maxOrdinal];
    }
```

```
36    }
37    // List Pods in descending order, since that determines
          rolling update order.
38    for (let ordinal of Object.keys(desiredPods).sort((a, b) => a
          - b).reverse()) {
39      desired.children.push(desiredPods[ordinal]);
40    }
41    if (catset.spec.volumeClaimTemplates) {
42      let desiredPVCs = {};
43      for (let template of catset.spec.volumeClaimTemplates) {
44        let baseName = `${template.metadata.name}-${catset.metadata
              .name}`;
45        for (let i = 0; i < catset.spec.replicas; i++) {
46          desiredPVCs[i] = newPVC(`${baseName}-${i}`, template);
47        }
48        // Also generate a desired state for existing PVCs outside
              the range.
49        // PVCs are retained after scale down, but are deleted with
               the CatSet.
50        if (observed.children && observed.children['
              PersistentVolumeClaim.v1']) {
51          for (let pvc of Object.values(observed.children['
                PersistentVolumeClaim.v1'])) {
52            if (pvc.metadata.name.startsWith(baseName)) {
53              let ordinal = getOrdinal(baseName, pvc.metadata.name);
54              if (ordinal >= catset.spec.replicas) desiredPVCs[
                    ordinal] = newPVC(pvc.metadata.name, template);
55            }
56          }
57        }
58      }
59      desired.children.push(...Object.values(desiredPVCs));
60    }
61    return desired;
62  },
63  };
```

Listing 5.8: catset-controller 的配置文件

```
---
apiVersion: universalcontroller .njuics .cn/v1alpha1
kind: UniversalController
metadata:
  name: catset- controller
spec:
  parentResource:
    apiVersion: mlhub.njuics .cn/v1alpha1
    resource: catsets
    revisionHistory:
      fieldPaths:
        - spec.template
  childResources:
    - apiVersion: v1
      resource: pods
      updateStrategy:
        method: RollingRecreate
        statusChecks:
          conditions:
            - type: Ready
              status: "True"
    - apiVersion: v1
      resource: persistentvolumeclaims
  hooks:
    sync:
      webhook:
        url: "http://catset-controller.universalcontroller:8080"
    finalize :
      webhook:
        url: "http://catset-controller.universalcontroller:8080"
```

Listing 5.9: CatSet 资源示例

```
apiVersion: mlhub.njuics .cn/v1alpha1
kind: CatSet
metadata:
  name: nginx-backend
```

```
spec:
  serviceName: nginx-backend
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
        component: backend
    spec:
      terminationGracePeriodSeconds: 1
      containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
      labels:
        app: nginx
        component: backend
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

YAML 文件 5.8是 catset-controller 的声明式定义。Yaml 文件 5.9是一个 CatSet 资源的示例，CatSet 的 spec 结构与 StatefulSet 的 spec 结构完全一样。

CatSet 的控制器依然用 JavaScript 实现，代码段 5.7是主逻辑，需要生成带

表 5-1: 四节点集群的服务器配置

硬件类型	型号	规格
CPU	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz	20 核
GPU	NVIDIA 1080Ti	2
内存	DDR4	128GB
网卡	Mellanox Technologies MT26448	10Gbps
磁盘	TOSHIBA MG04SCA20EN	2TB

编号的 Pods 和 PVC（如果有的话），为了支持滚动更新，还要把 Pods 按照编号排好序再返回，因为滚动更新是按照调谐结果中子资源列表中的顺序逐个更新的。

5.3.3 对比总结

CatSet 重新实现了 StatefulSet，并且支持滚动更新。为了支持滚动更新，开发者只需要编写 YAML 文件 5.8的第 16 到第 21 行，为 pods 子资源加上滚动更新策略，以及编写代码 5.7的第 38 行到第 40 行将期望存在的 Pods 按照序号降序排列即可。开发者总共只添加了 9 行代码就让 CatSet 支持了滚动更新，而为了让 StatefulSet 支持滚动更新，Kubernetes 开发者改动了业务逻辑、模版文件、生成的代码，总共涉及到了超过 9000 行的改动[13]。UC 提供的声明式接口帮助开发者快速地使自己的应用支持滚动更新。

5.4 性能测试

Operator 一般都不是计算型任务，运行时 CPU 的占用率极低，特别是在（超）小型集群中，接近于 0。但是因为要监视资源，也就是订阅并且建立缓存，内存开销和网络开销呈线性增长。所以性能测试主要从这两个维度进行分析。

5.4.1 实验环境

在表 5-1所描述的集群上搭建了 Kubernetes 集群用于实验。

5.4.2 对比方法

我设计了六种场景，用于论证相比于部署多个 Operators，使用 Universal-Controller 可以消耗更少的内存和网络带宽。每个场景准备前都要清理环境，重新安装 Kubernetes，之后部署 100 个只执行“sleep 365d”的 Pod 当做负载。我还实现了一个 donothing-operator，它的 CRD 为 DoNothing，会订阅 Pod 和 PVC，但是什么都不干。实现它的目的是为了与 CatSet 对比，CatSet 也会订阅 Pod 和 PVC。

接下来每个场景会安装不同的 Operators：

- **场景 1**：不安装任何 Operator。
- **场景 2**：只安装 UniversalController。
- **场景 3**：只安装原版的 tf-operator。
- **场景 4**：先安装 UniversalController，再安装重新实现的 tf-operator。
- **场景 5**：安装 donothing-operator 和原版的 tf-operator。
- **场景 6**：先安装 UniversalController，再安装重新实现的 tf-operator 以及 catset-operator。

一个 Opeator 在安装之后，首先会与 api-server 进行同步，建立它关心的资源的缓冲，所有启动之后会有一小段网络流量高峰，之后回落，再趋于平稳，内存也是先快速增长，之后趋于平稳。我会将每个场景中 api-server 在 Operators 启动后的前 5 分钟内上传的总数据量作为网络负载参考量，将之后 5 分钟内 Operators 所占用的总内存的平均值作为内存负载参考量。

5.4.3 实验结果

表 5-2汇总了各个场景的结果。场景 1 中没有任何 Operator，但是每个节点的 Kubelet 都需要与 api-server 同步信息，所有也有很多数据需要传输。场景 2 中安装了 UniversalController，但是 UC 它只会监听 UC CRD，集群中没有任何 UC CRD，所以 api-server 的网络负载几乎不变。场景 3 中 tf-operator 需要订阅 TFJob、Pod、Service，所以 api-server 的网络负载增加了不少。

对比场景 3 和场景 4 可以看到，因为 UC Controller 自身带来的负载，重新实现的 tf-operator 的内存和网络负载都要比原版的 tf-operator 要高。

但是对比场景 5 和场景 6 可以看到，再增加一个 operator 后，场景 5 的内存和网络负载要更高，比场景 3 增长很多，而场景 6 与场景 4 的负载很接

表 5-2: 性能测试

场景编号	Operators 内存总用量（MB）	5 分钟内上传数据量（KB）
1	0	11995.14
2	12.52	12012.05
3	13.87	13297.24
4	19.18	13438.46
5	25.78	14529.64
6	21.93	13542.36

近。tf-operator 和 catset-operator 都需要订阅 Pod 资源，但是当它们都部署在 UniversalController 之上时，Pod 资源只会被订阅一次，这部分就不会带来额外的负载，catset-operator 还需要订阅 Service 和 CatSet，但是我们当前集群中主要的资源都是 Pod，Service 很少，还没有 CatSet，所以也没有产生很多负载。

借助 UniversalController 的共享信息器（SharedInformer），当部署多个 Operators 时，部署在 UniversalController 之上要比每个单独部署占用更少的内存和网络带宽。

5.5 小结

本章设计了多个用例，用 UniversalController 实现了三个功能各异的 Kubernetes Operators，验证了 UniversalController 可以简化 Operator 的实现，并且有很强的通用性。

本章设计的性能测试也证实 UniversalController 借助于共享通知者（SharedInformer），避免了重复订阅同一个资源，相比于一般的多控制器部署方式对内存和网络的占用更小。

第六章　总结和展望

6.1　工作总结

随着云计算的蓬勃发展，新技术不断涌现。Docker 和 Kubernetes 的出现更是重要的里程碑。Kubernetes 已经成为了容器编排的实时标准，是云计算重要的基础设施。但是 Kubernetes 提供的现有 APIs 不一定能够很好的满足使用者的需求，使用者经常需要去扩展 Kubernetes 以更好的支持自己的应用的部署、更新和维护。最主流的 Kubernetes 扩展方式就是 Kubernetes Operators，大量的 Operators 开始在开源社区出现。然而，编写一个 Operator 并不容易，具有相当高的门槛，并且需要付出大量的精力和时间。Operator 开发人员需要一定程度的 Kubernetes 和分布式系统知识，需要写大量的模版代码或者使用代码生成工具，编写出的 Operator 帮助我们实现了应用程序的自动化运维，但是维护这个 Operator 却还是要给开发人员带来很大的负担。

本文提出了一种声明式的通用 Kubernetes Operator，为用户开发 Operator 提供一种简单的新方式，让用户摆脱 Go 语言、Kubernetes 开发工具包、代码生成工具的学习与使用成本，用更加声明式的方式开发 Operator，将注意力完全集中在核心业务逻辑上，并且可以使用任意自己喜欢或熟悉的语言来实现一个标准优质的 Operator。本文将该工具成为 UniversalController，它自身也是一个 Operator，底层实现是经典的控制器模式，但是把业务逻辑部分抽取出来托管给用户编写的 hooks。

借助 UniversalController 提供的声明式 API，用户在写核心业务逻辑时也可以获得平时使用 yaml 编写配置文件并使用 kubectl apply 部署相近的体验，只是需要改用 json 编写一些配置文件。如果用户已经很熟悉用 kubectl apply 去使用 Kubernetes 的声明式 API 来管理应用，那么就可以很容易地基于 UniversalController 实现一个 Operator 为应用的部署、更新、维护提供自动化流程而不必去学习 Go 语言或者如何使用 Kubernetes 客户端库，也不需要去学习使用代码生成工具。

6.2 未来展望

本文提出的工作将 Kubernetes 操作相关的代码从业务逻辑中提取了出来，用户不用再关注 Kubernetes 的 Client API，让用户将开发工作集中在业务逻辑上，帮助用户减少了大量的开发工作。同时，本文仍然存在需要在未来工作中进行改进的地方。

本文提出的工作让用户将开发工作集中在业务逻辑上，但是用户必须借助 serverless 工具或者自己编写网络处理相关代码来启动一个 web 服务，以便与 UniversalController 对接。未来的工作中会加入更多的机制，例如 gRPC 或者嵌入式的脚本代码，让用户可以有更多的选择。或者可以将 UniversalController 的一部分封装成更加通用的库，提供一些方便的开发接口，开发者可以将业务逻辑实现成系统内部的代码调用，这样就不用将业务逻辑放在 UniversalController 的外部组件内，省去网络通信的开销。

参考文献

[1] PISCAER J. The Gorilla Guide to Kubernetes in the Enterprise[M]. [S.l.]: ActualTech Media in collaboration with Platform9, 2019.

[2] The Kubernetes Authors. kubernetes/client-go[EB/OL]. [2021-04-15]. https://github.com/kubernetes/client-go.

[3] PAHL C, BROGI A, SOLDANI J, et al. Cloud Container Technologies: A State-of-the-Art Review[J/OL]. IEEE Transactions on Cloud Computing, 2019, 7(3): 677 – 692. http://dx.doi.org/10.1109/TCC.2017.2702586.

[4] The Docker Company. Docker and red hat announce major alliance[EB/OL]. 2014 (2014/4/15) [2021/4/15]. https://www.redhat.com/zh/about/press-releases/docker-and-red-hat-expand-collaboration-around-container-technologies.

[5] Joe Fernandes. OpenShift and Kubernetes: Where We've Been and Where We're Going Part 1[EB/OL]. [2021-04-15]. https://www.openshift.com/blog/openshift-kubernetes-where-weve-been-and-where-were-going-part-1.

[6] The Kubernetes Authors. What is Kubernetes[EB/OL]. The Linux Foundation, 2021 (2021/2/1) [2021/4/15]. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[7] Cade Metz. Google open sources its secret weapon in cloud computing[EB/OL]. 2014 (2014/6/10) [2021/4/15]. https://www.wired.com/2014/06/google-kubernetes/.

[8] MCLUCKIE C. From Google to the world: the Kubernetes origin story[J]. Google Cloud Blog, 2016.

[9] The Kubernetes Authors. Kubernetes Concepts[M]. [S.l.]: The Linux Foundation, 2020.

[10] The Kubernetes Authors. Set up High-Availability Kubernetes Masters[EB/OL]. The Linux Foundation, 2020 (2020/11/19) [2021/4/15]. https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/.

[11] ELLINGWOOD J. An Introduction to Kubernetes[EB/OL]. [2021-04-15]. https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes.

[12] The Kubernetes Authors. Operator pattern[EB/OL]. The Linux Foundation, 2021 (2021/2/23) [2021/4/15]. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[13] GitHub User Kenneth Owens. implements StatefulSet update[EB/OL]. 2017 (2017/6/7) [2021/4/15]. https://github.com/kubernetes/kubernetes/pull/46669.

[14] MORRIS K. Infrastructure as Code: Managing Servers in the Cloud[M/OL]. [S.l.]: O'Reilly Media, 2016. https://books.google.de/books?id=kOnurQEACAAJ.

[15] Tung Nguyen. Kustomize vs Helm vs Kubes: Kubernetes Deploy Tools[EB/OL]. 2020 (2020/11/5) [2021/4/15]. https://blog.boltops.com/2020/11/05/kustomize-vs-helm-vs-kubes-kubernetes-deploy-tools.

[16] The KUDO Authors. KUDO vs Custom Controllers[EB/OL]. [2021/4/15]. https://kudo.dev/docs/comparison/custom-controllers.html.

[17] The Kubernetes Authors. Controllers[EB/OL]. The Linux Foundation, 2021 (2021/2/3) [2021/4/15]. https://kubernetes.io/docs/concepts/architecture/controller/.

[18] The Kubernetes Authors. Kubernetes components[EB/OL]. The Linux Foundation, 2021 (2021/3/18) [2021/4/15]. https://kubernetes.io/docs/concepts/overview/components/.

[19] Glenn Berry. Scaling sql server 2012[EB/OL]. [2021/4/15]. http://www.pass.org/eventdownload.aspx?suid=1902.

[20] SOSINSKY B. Cloud Computing Bible[M]. 1st. [S.l.]: Wiley Publishing, 2011.

[21] MELL P, GRANCE T, OTHERS. The NIST definition of cloud computing[J], 2011.

[22] Mary Branscombe. The Runaway Problem of Kubernetes Operators and Dependency Lifecycles[EB/OL]. 2020 (2020/8/18) [2021/4/15]. https://thenewstack.io/the-runaway-problem-of-kubernetes-operators-and-dependency-lifecycles/.

[23] coreos.com. Kubernetes operators[EB/OL]. 2020 [2021/4/15]. https://coreos.com/operators/.

[24] DOBIES J, WOOD J. Kubernetes Operators: Automating the Container Orchestration Platform[M/OL]. [S.l.]: O'Reilly Media, 2020. https://books.google.com/books?id=Kf3RDwAAQBAJ.

[25] FERNANDEZ T. What is Infrastructure as Code[EB/OL]. [2021-04-15]. https://blog.stackpath.com/infrastructure-as-code-explainer/.

[26] Simon Harrer Florian Beetz, Anja Kammer. gitops is continuous deployment for cloud native applications[EB/OL]. [2021-04-15]. https://www.gitops.tech/.

[27] Alexis Richardson. Gitops - operations by pull request[EB/OL]. [2021-04-15]. https://www.weave.works/blog/gitops-operations-by-pull-request.

[28] Controller Runtime Documentation[EB/OL]. [2021-04-15]. https://godoc.org/github.com/kubernetes-sigs/controller-runtime/pkg.

[29] github.com/kubernetes-sigs. Repo for the controller-runtime subproject of kubebuilder (sig-apimachinery)[EB]. .

[30] Bobby Tables. Stay informed with kubernetes informers[EB/OL]. [2021-04-15]. https://www.firehydrant.io/blog/stay-informed-with-kubernetes-informers/.

[31] AUTHORS T K. client-go under the hood[EB/OL]. [2021-04-15]. https://github.com/kubernetes/sample-controller/blob/master/docs/controller-client-go.md.

[32] ENDRES C, BREITENBÜCHER U, FALKENTHAL M, et al. Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications[C] // . 2017.

[33] TIOBE Software BV. TIOBE Index | TIOBE - The Software Quality Company[EB/OL]. [2021-04-15]. https://www.tiobe.com/tiobe-index/.

[34] HYKES S. Lightning talk - the future of linux containers[J]. PyCon, 2013.

[35] SPAZZOLI R. Kubernetes Operators Best Practices[EB/OL]. [2021-04-15]. https://www.openshift.com/blog/kubernetes-operators-best-practices.

[36] Alex Giamas. From Monolith to Microservices, Zalando's Journey[EB/OL]. 2016 (2016/2/11) [2021/4/15]. https://www.infoq.com/news/2016/02/Monolith-Microservices-Zalando/.

[37] Tony Mauro. Adopting Microservices at Netflix: Lessons for Architectural Design[EB/OL]. 2015 (2015/2/19) [2021/4/15]. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[38] Red Hat Press Office. Red Hat to Acquire CoreOS, Expanding its Kubernetes and Containers Leadership[EB/OL]. 2018 (2018/1/30) [2021/4/15]. https://www.redhat.com/en/about/press-releases/red-hat-acquire-coreos-expanding-its-kubernetes-and-containers-leadership.

[39] Brandon Philips. Introducing the Operator Framework: Building Apps on Kubernetes[EB/OL]. 2018 (2018/5/1) [2021/4/15]. https://www.redhat.com/en/blog/introducing-operator-framework-building-apps-kubernetes.

[40] TURIN G, BORGARELLI A, DONETTI S, et al. A Formal Model of the Kubernetes Container Framework[C/OL] // . 2020.
http://dx.doi.org/10.1007/978-3-030-61362-4_32.

[41] BERNSTEIN D. Containers and Cloud: From LXC to Docker to Kubernetes[J/OL]. IEEE Cloud Computing, 2014, 1(3): 81 – 84.
http://dx.doi.org/10.1109/MCC.2014.51.

[42] LUKA M. Kubernetes in Action: Anwendungen in Kubernetes-Clustern bereitstellen und verwalten[C] // . 2018.

[43] BREWER E A. Kubernetes and the Path to Cloud Native[C/OL] // SoCC '15: Proceedings of the Sixth ACM Symposium on Cloud Computing. New York, NY, USA: Association for Computing Machinery, 2015: 167.
https://doi.org/10.1145/2806777.2809955.

[44] ANON. Extending Kubernetes with the Operator Pattern[C]. Portland, OR: USENIX Association, 2019.

[45] IBRYAM B, HUSS R. Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications[M/OL]. [S.l.]: O'Reilly Media, 2019.
https://books.google.com.tw/books?id=8WmRDwAAQBAJ.

[46] HAUSENBLAS M, SCHIMANSKI S. Programming Kubernetes: Developing Cloud-Native Applications[M/OL]. [S.l.]: O'Reilly Media, 2019.
https://books.google.com/books?id=7VKjDwAAQBAJ.

[47] BURNS B, VILLALBA E, STREBEL D, et al. Kubernetes Best Practices: Blueprints for Building Successful Applications on Kubernetes[M/OL]. [S.l.]: O'Reilly Media, 2019.
https://books.google.com/books?id=Cju-DwAAQBAJ.

[48] FLEMING S. Kubernetes Handbook: Non-Programmer's Guide To Deploy Applications With Kubernetes[M/OL]. [S.l.]: CreateSpace Independent Publishing Platform, 2018.
https://books.google.com/books?id=Z23RugEACAAJ.

[49] SUTTER B, SAMPATH K. Knative Cookbook: Building Effective Serverless Applications with Kubernetes and OpenShift[M/OL]. [S.l.]: O'Reilly Media, 2020.
https://books.google.com/books?id=RIziDwAAQBAJ.

[50] DOBIES J, WOOD J. Operadores do Kubernetes: Automatizando a plataforma de orquestração de contêineres[M/OL]. [S.l.]: Novatec Editora, 2020.
https://books.google.com/books?id=HnjpDwAAQBAJ.

[51] RAUL A. Cloud Native with Kubernetes: Deploy, configure, and run modern cloud native applications on Kubernetes[M/OL]. [S.l.]: Packt Publishing, 2021.
https://books.google.com/books?id=omYNEAAAQBAJ.

学位论文出版授权书

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：_____

_____年_____月_____日

论文题名	声明式的通用 Kubernetes Operator 的设计与实现				
研究生学号		所在院系	计算机科学与技术系	学位年度	XXXX
论文级别	☑ 硕士 ☐ 硕士专业学位 ☐ 博士 ☐ 博士专业学位 （请在方框内画勾）				
作者电话			作者 Email		
第一导师姓名			导师电话		

论文涉密情况：

☐ 不保密

☑ 保密，保密期：_____年_____月_____日 至 _____年_____月_____日

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。