南京大学

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 汪浩港

学科、专业方向 计算机科学与技术

研 究 方 向 软件方法学

指 导 教 师 曹春 教授

2021 年 **5** 月 **17** 日

学　　号：**MG1833067**

论文答辩日期：**2021** 年 **6** 月 **1** 日

指 导 教 师：　　（签字）

The Design and Implementation of A Declarative Universal Kubernetes Operator

by

Wang Haogang

Supervised by

Professor Cao Chun

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

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

计算机科学与技术 专业 2018 级硕士生姓名：汪浩港

指导教师（姓名、职称）：曹春 教授

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 的声明式 API 的最主流的方式。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，从而更简单地开发和部署自定义控制器。具体而言，本文工作的主要内容包括：

1. 针对 Operator 开发困难的问题，提出一种声明式的通用调谐技术，简化需要编写的代码，大量减少 Operator 开发者的工作量，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，即描述期望状态上。
2. 实现了声明式的通用 Kubernetes Operator，UC。该工具具有声明式的资源监视（watch），声明式的调谐、声明式的更新策略和语言无关的特性。用户不需要编写任何与 Kubernetes 交互的代码，只需要在 YAML 文件中描述需要监听的资源、使用的更新策略以及在调谐代码段中描述期望的状态即可。
3. 基于 UC 重新实现了一些现有的 Operator，证明了 UniversalController 可以极大的缩减开发工作量，并且适用于大部分场景的开发。同时性能测试验证了它还能在多自定义控制器部署的环境中减少内存消耗和 kube-apiserver 的负载。

关键词：Kubernetes；Operator；声明式；UniversalController

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 Declarative Universal Kubernetes Operator

SPECIALIZATION: Computer Science and Technology

POSTGRADUATE: Wang Haogang

MENTOR: Professor Cao Chun

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 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 is helpful for developing and deploying custom controllers.Specifically, the main elements of the work in this paper include:

1. To address the problem of difficult Operator development, a declarative universal reconciliation technique is proposed to simplify the code that needs to be written, to massively reduce the workload of Operator developers, to eliminate the burden of learning the Kubernetes client library, the Kubernetes API mechanism library or other tools, and to not have to write or generate template code, but to focus on business logic, i.e., describing the desired state.
2. Implements a declarative universal Kubernetes Operator, named as UniversalController, with declarative resource watch, declarative reconciliation, declarative update policies, and language-agnostic features. Instead of writing any code to interact with Kubernetes, users only need to describe the resources to be listened to, the up-

date policies to be used in a YAML file and the desired state in the reconciliation snippet.

3. Based on UniversalController, some existing Operators were re-implemented, proving that UniversalController can greatly reduce development effort and is suitable for most scenarios. Performance testing also verified that it can reduce memory consumption and kube-apiserver load in environments with multiple custom controller deployments.

keywords: Kubernetes; Operator; Declarative; UniversalController

目　　次

插图清单

附表清单

第一章　绪论

1.1　研究背景

在过去十年中，软件和 IT 系统的开发、托管、交付和扩展方式发生了根本性的转变。大规模分布式应用不断产生，新的需求也不断产生，集群管理人员希望应用能够快速适应用户流量的波动，同时最大限度地降低 IT 基础设施的管理成本，这些都为云技术被广泛采用铺平了道路。正如 Netflix 或 Zalando 等颇具规模的互联网科技公司所展示的那样，软件开发的方式和方法已经从每年几次的大型单体应用的计划驱动交付转变为由数百个单一用途的服务组成的系统，每个服务都是独立和可多次部署的，一个服务往往一天内就有多个版本的迭代[2][3]。

尽管云计算具有相当的灵活性，但随着应用和平台的复杂性增加，仍然需要对基础应用、基础设施和相关流程进行创新与改进。应用程序和基础设施运营团队在管理基础设施方面正面临着挑战，这些基础设施需要同时支持数百甚至数千的应用程序。在这种情况下，传统的自动化方法，如使用特制的、指令式的脚本，被证明是难以管理和扩展的。相反，最近越来越流行的自动化工具旨在遵循基础设施即代码（Infrastructure as code，IaC）原则。根据这一原则，基础设施和核心服务的整体配置和状态要用（典型的声明性）代码来定义。然后，借助相关工具，这个刻画了期望状态定义的代码，可以自动转换为正确的指令和 API 调用，从而产生完全符合期望状态的配置资源。这种将期望状态的定义与实际状态同步的过程被称为 “状态调谐”，一些现代的基础设施和云计算自动化工具都采用了这种方法，其中最受欢迎的是 Terraform 和基于容器的平台 Kubernetes。

Kubernetes 已经开始被广泛应用于基础设施自动化和状态调谐的使用场景。它是目前最受欢迎的管理基于容器的应用程序的平台，它从一开始就围绕着状态调谐的概念设计。整个架构可以被认为是一个各个模块通过共享状态存储协作的调谐循环系统。此外，可扩展性是 Kubernetes 设计中的一个核心方面。通过结合这两个特点，Kubernetes 也被用作一个通用的状态调谐引擎，能

够调谐定制的、特定于应用程序的资源和状态，这些资源和状态甚至可以是Kubernetes 本身的外部资源，如云提供商资源或虚拟数据中心设备。为了利用这些功能，应用程序需要正确有效地处理与 Kubernetes API 的通信和集成，这些并不是简单的工作。由于 Kubernetes 仍然是一个相对年轻的项目，试图解决这种复杂性的可用库非常少。Kubernetes 本身与许多其他容器和云原生技术类似，是用 Go 语言编写的，所以最成熟的集成和扩展库也是用 Go 语言编写的。也就是说，只要相关的代码都可以用 Go 语言编写，开发体验会是最好的。这对于一部分开发者和公司是可能的，但大部分其他语言的开发者将不得不放弃许多已有的标准和工具。更先进、支持更多编程语言的 Kubernetes 抽象库才会让更多人收益。

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[4]。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 开发人员需要具备一定程度的 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[5]。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，可以有效减轻开发人员的开发与运维负担。

1.2 研究现状

在开发 Operator 时，最常用且灵活的做法是使用现有的 Kubernetes 客户端，可以是 Go、Java、JS/Typescript 或其他语言的客户端。这些客户端提供了对 Kubernetes API 的直接底层访问，没有任何包装或附加层。其中最成熟且功能齐全的是 Go 客户端 client-go[①]，它提供了很多基础组件用于自动以控制器的开发。但是 client-go 不是一个专门用于实现控制器的库，而是为了更通用的场景而设计的，用它来实现 Operator 还是需要接触到太多的底层接口，十分繁琐。而如果用 Go 以外的编程语言，体验更是会大幅下滑，它们没有 client-go

[①]https://github.com/kubernetes/client-go

成熟，甚至缺少很多特性。

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[①]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 提高了开发人员使用 Go 语言快速构建和发布 Kubernetes API 的速度并降低了管理的复杂性。它们都使用了高级抽象库 controller-runtime[②]，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，给出了脚手架，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[5]。

对于其他编程语言的使用者，Operaoter 的开发体验很不友好，没有类似 controller-runtime 的高级抽象包，甚至官方提供的客户端都还不够成熟。

1.3 本文工作

本文针对现有 Operator 开发方式中存在的各种问题，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，用声明式的方式开发 Operator。自定义资源和自定义控制器都借助 Kubernetes 的声明式 api 创建，用户可以将注意力完全集中在核心调谐逻辑上，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。本文将该工具称为 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核心调谐逻辑代码（hooks）。

借助 UniversalController 提供的声明式 API，尤其是声明式调谐接口，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

[①]https://github.com/kubernetes-sigs
[②]https://github.com/kubernetes-sigs/controller-runtime

apply” 命令部署相近的体验，只是需要改用 JSON[①]编写一些配置文件，并且可以使用任意自己熟悉或者喜欢的编程语言来实现。如果用户已经很熟悉用 “kubectl apply” 命令去使用 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。本文工作主要包括：

1. 针对 Operator 开发困难的问题，提出一种声明式的通用调谐技术，简化需要编写的代码，大量减少 Operator 开发者的工作量，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，即描述期望状态上。
2. 实现了声明式的通用 Kubernetes Operator，UniverslController。该工具具有声明式的资源监视（watch），声明式的调谐、声明式的更新策略和语言无关的特性。用户不需要编写任何与 Kubernetes 交互的代码，只需要在 YAML 文件中描述需要监听的资源、使用的更新策略以及在调谐代码段中描述期望的状态即可。
3. 基于 UniversalController 重新实现了一些现有的 Operator，证明了 Universal-Controller 可以极大的缩减开发工作量，并且适用于大部分场景的开发。同时性能测试验证了它还能在多自定义控制器部署的环境中减少内存消耗和 kube-apiserver 的负载。

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 存在的问题以及本文针对这些问题所做的工作做了简单的介绍。

第 2章相关工作和技术。本章主要介绍 Kubernetes Operator 所涉及到的关键技术与工作。首先介绍了现代基础设施的特征和面临的挑战，之后深入到一个具体的、流行的做法，基础设施即代码，然后给出状态调谐的定义和细节，最后深入探讨了 Kubernetes 系统和它如何应用和实现状态调谐的细节。

第 3章声明式的通用调谐技术。本章首先对现有的 Operator 开发方式进行

[①]https://www.json.org/json-en.html

了分析，指出问题所在。然后解释 UniversalController 如何通过声明式的通用调谐技术解决这些问题。

第4章声明式的通用 Kubernetes Operator 的设计与实现。本章详细描述本文提出的 UniversalController 的设计思想与具体实现。

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

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　动态基础设施

自诞生以来，信息技术行业及其相关应用经历了快速和彻底的演变。在过去，大多数软件服务都与基础设施高度耦合，而基础设施是由一组物理硬件设备组成的，如带有专用硬盘和网络接口及设备的裸金属服务器。

最初提供和操作这样的基础设施需要大量的计划和人工工作，需要对每个设备和装置单独进行仔细的安装和配置。典型的情况是，这种配置是高度定制的，专门为它计划承载的软件工作负载而定制[6]。换句话说，如微软公司杰出工程师比尔贝克（Bill Baker）最初使用的流行比喻[7] 所说，服务器和基础设施组件在过去被视为独特的、不可缺少的、人工 “养育” 和单独照顾的 “宠物”。

如今，底层的物理硬件往往与应用程序和工作负载更加脱钩，中间有多个抽象层。现代应用背后的基础设施、平台和服务是高度动态和自动化的。软件通常每天都会被多次部署到选定的虚拟化计算资源中，如虚拟机或容器。这些资源是同质的和通用的[6]。配置和维护是自动处理的，而不需要管理员的亲身参与。当一个服务器出现问题时，不需要试图去修复它。相反，有问题的服务器被销毁，一个新的、相同的服务器被启动。同质性允许应用程序通过添加更多的服务器实例来扩展，而不是向专用服务器添加资源（扩大规模）[6]。

在比尔贝克的比喻中，今天的 IT 系统被认为是 “牛”，可以作为一个群体来管理，就所有的意图和目的而言，彼此都是相同的，而且很容易被替换。在本文中，我们将使用 Kief Morris 在《Infrastructure as Code》一书中使用的术语动态基础设施（dynamic infrastructure）来指代这种基础设施[6]。

2.1.1　特征

动态基础设施的概念可以通过提供一组特征来进一步定义，这些特征在许多甚至所有的动态基础设施供应商和系统中应该是一致的。

动态基础设施这一术语旨在进一步概括类似云的平台形式。Morris 定义了一种普遍的特征，指出一个动态的基础设施平台必须是可编程的、按需的和自

我服务的[6]。

可编程

一个动态的基础设施平台需要是可编程的，这意味着它需要使无头（headless）软件和脚本能够使用远程 API 与它进行编程式的互动，并附带一套软件库或开发工具包[6]。它强调了使用标准协议来实现这种 API。

按需

与 NIST 的云计算特征不同，Morris 在按需和自助服务方面认识到更多的细微差别，并将它们区分开。与 NIST 的定义相似，按需这个要求表达了对动态基础设施平台的需求，允许立即创建和销毁资源，而不需要求助于昂贵和冗长的过程，如服务票据[6]。

自助服务

自助服务的要求是对按需服务的延伸，强调在易于创建和销毁资源的基础上，还需要能够让平台的用户高度定制资源。用户应该能够使用该平台来完全定制相关的资源，以满足他们的具体使用情况[6]。

NIST 提出的云计算的特征，以及 Morris 对动态基础设施平台的要求，都说明了现代基础设施的性质。就本文而言，Morris 的三个要求为从用户（或团队）的角度谈论与自动化和管理有关的动态基础设施提供了一个更合适的框架。

2.2 基础设施即代码

在上一节描述了动态基础设施的特性之后，本节将介绍被称为基础设施即代码的做法。这种做法代表了有效利用动态基础设施的潜力以及解决其复杂性和挑战的一种可行的、流行的方法。

现代基础设施的生命周期正变得与软件应用程序越来越相似。现代基础设施的组件更加抽象。可以根据需要立即配置和改变它们，这意味着迭代和改变的速度也在增加。

Morris 将基础设施即代码（IaC）定义为一种基于软体开发实践的基础设施自动化方法，重点在于提供和改变系统及其配置的一致、可重复的程序[6]。

使用 IaC，基础设施的每个方面都在一个或多个文件中使用某种形式的代码来定义。有了这个规则，就可以设计出利用自动化工具的流程，以便根据代码文件中定义的规范，自动提供资源或对基础设施进行修改。

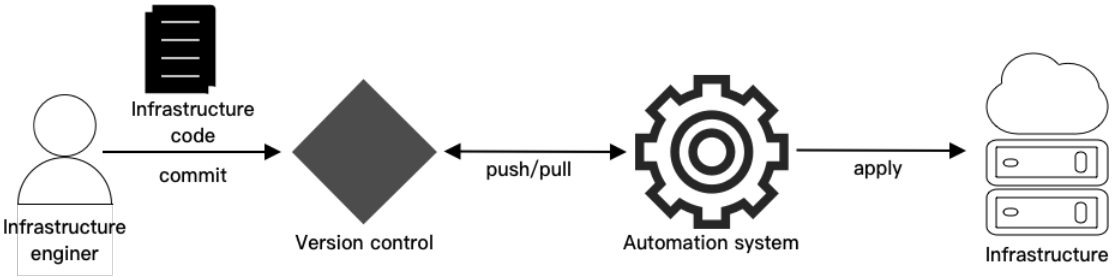

图 2-1: 基础设施即代码的工作流程

在一个典型的基础设施即代码的工作流程中，如图2-1所示，为了进行一次变更，基础设施工程师会在一个包含基础设施代码的文件中表达这个变更。与应用程序代码类似，该代码可以被提交到版本控制系统的存储库中。之后，代码被推送到自动化系统或由自动化系统拉出，随后自动化系统使用该文件和平台特定的集成功能，以便在基础设施中应用所描述的变更[8]。

2.2.1 核心做法

基础设施即代码的方法囊括了受软件开发启发的若干惯例做法。本节将介绍在本文中被认为是基础的三种惯例做法：定义文件、自文档和版本控制，并讨论它们的好处。

2.2.1.1 定义文件

使用定义文件来描述基础设施是基础设施即代码的核心。按照更传统的方法，基础设施资源通常是用自动化系统的图形界面来定义，并存储在其数据库中，或者根本没有严格的定义，只是在图表和规范文件中记录。按照基础设施即代码的方法，基础设施的所有方面和资源都被定义为代码，也就是文本文件[6]。

使用定义文件有几个好处。首先，它们允许精确和详细地描述基础设施资源。此外，对定义的修改可以通过一个文本编辑器来完成，这通常比使用图形界面更快、更简单。最后，定义文件有助于使基础设施更加一致和可重复使用，因为文本定义可以通过最小的调整来复制以适应新的使用情况。另外，根

据所使用的 IaC 工具和语言，可以使用更高层次的编程结构，如模板和函数，以进一步简化这一过程[6][8]。定义文件的使用直接促成了其余两种惯例做法。

2.2.1.2 自文档

自文档可以被认为是一种惯例做法，也是使用基础设施即代码和定义文件可以得到的直接好处。在传统的方法中，变化的实施和文档是两个独立的过程。这往往会导致文档过时或不存在，因为随着频繁的变化，要保持文档的更新是一个挑战。通过使用精确和详细的代码在定义文件中定义基础设施，代码和文件会自动触发相应行为，并可作为基础设施的文档[6][8]。

2.2.1.3 版本控制

使用代码和文件来描述基础设施，是软件工程中最广泛使用的做法之一，即版本控制，能够被用于基础设施。使用像 Git 这样的版本控制系统（VCS），代码可以在代码仓库中进行组织和版本管理。对代码所做的每一个改动都必须提交到版本控制系统中，版本控制系统会跟踪所有改动的历史，并支持在历史的不同点上查看代码库的不同状态（快照）。因此，代码仓库可以作为代码的可信来源。

这可以为基础设施代码服务。首先，对所有基础设施的定义有一个单一的可信来源，可以改善协作和变化的一般可见性，因为历史日志可以作为一个易于使用的时序的变化描述。此外，这也允许可追溯性和可审计性，因为每一个变更都可以追溯到 VCS 中的提交，而提交中通常都有关于做出变更的人的信息，也有关于变更的描述或理由。最后，历史日志和 VCS 的操作也可以在回滚时有所帮助，基础设施可以恢复到一些经过测试的安全状态。

2.2.2 工具和范式

为了根据定义文件正确和有效地管理实际的基础设施，需要自动化工具或系统。本节介绍基础设施即代码工具的概况，并讨论了所使用的两种主要编程范式。

2.2.2.1 工具类型

目前 IaC 和相关工具的前景相当广阔，我们可以将它们大致分为以下四类[8]。

脚本工具

使用常见的操作系统脚本工具和语言，如 Bash、Powershell、Python、Perl 等，是使用代码管理基础设施的最简单方法。虽然它们足以完成简单的任务，但这些工具在更复杂的情况下不能很好地扩展。这些工具本质上使用的是指令式的方法。

配置惯例系统

配置惯例系统是一套功能更全面的系统，如 Chef、Puppet 或 Ansible，它们通常用于以通用方式管理服务器。这些工具倾向于使用标准或专门的远程连接协议和代理与服务器直接通信，以便对软件进行安装和配置。一般来说，这类工具倾向于使用特定的概念和术语，指令式和声明式的工具都可以找到。

配置（provisioning）工具

这类工具通常提供更高层次的抽象，允许用户创建、修改和删除动态基础设施平台的资源。其中最知名的例子是 AWS CloudFormation 和 Terraform。两者都采用声明式方法，其中 CloudFormation 使用 JSON，Terraform 使用自定义的、特定领域的配置语言（DSL）来定义文件。CloudFormation 是专门针对 AWS 平台的，而 Terraform 则可以在许多不同的平台上使用，因为它可以通过定制的供应商来扩展，几乎可以用于任何符合动态基础设施要求的平台。另外，大多数动态平台也以命令行工具或软件库的形式提供指令式配置工具，如 AWS 或 Google SDK。

基于容器的编排系统

基于容器的编排系统通常是以 IaC 原则设计的全动态基础设施平台。这方面的例子有 Docker Swarm、Kubernetes、Nomad 等等。其核心是提供一个基础设施级别的抽象，如计算、存储和网络。然而，容器的内在特征使 IaC 成为可能。容器封装了单个应用环境的全部内容，可以用代码（如 Dockerfile）

来定义，并打包成一个不可变的镜像。此外，容器编排系统（如 Kubernetes、Nomad）通过允许系统中的所有资源被声明性地定义，进一步拥抱 IaC。

2.2.2.2 指令式和声明式

IaC 工具和系统通常使用两种主要的编程范式：指令式和声明式。

使用指令式范式（如脚本或 Chef），定义文件中的代码本质上是一组指令，按照特定的顺序执行，以达到基础设施的期望状态。换句话说，指令式用户不仅要求用户描述所需的基础设施资源，而且至少在某种程度上还要求用户描述如何以及以何种顺序应用配置。

使用声明式方法，代码更简单，专门用于定义资源对象、列表和层次结构。在这种情况下，用户不需要知道在一个特定的平台上应该如何配置资源，甚至往往不需要知道以何种顺序配置。这些问题被抽象出来，委托给 IaC 工具本身。

这两种风格都有优点和缺点。为了强调要执行的单个指令，指令式范式可以被认为是一种更强大的方法，可以支持最复杂的特殊配置。另一方面，声明式方法不那么强大，而且通常受到所用 IaC 系统能力的限制。然而，从用户的角度来看，声明式编写的文件提供了更简单的读写体验，因为它不要求用户对如何改变基础设施有专业的知识。此外，它们还允许更快地发现和分析基础设施（文件）的不同状态之间的差异，并减少了工作量。

指令式工具使用时拥有许多专门的和不一致的配置，更像是把基础设施当作“宠物”而不是“牛”，使管理更具挑战性和不可扩展性，这种情况应该尽可能避免。这使得指令式工具的价值主张与声明式工具相比要逊色得多，声明式工具欢迎“牛”的方式，并促进标准化、一致性和可重复使用。

2.3 状态调谐

在这一节中，我们将探讨一种在大多数声明式 IaC 工具和系统中常见的模式，并从一般的角度讨论其变化和用途。这种模式被称为状态（state）调谐。

2.3.1 从实际状态到期望状态

声明式 IaC 工具的基本特征之一是能够接受对期望状态（如特定的基础设施资源集）的描述，并自动变更实际状态（如 AWS 账户中配置的对象/服

务），使其反映期望状态。

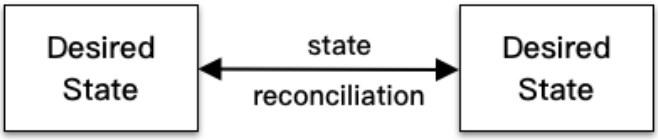

图 2-2: 状态调谐

图2-2中的这种模式就是状态调谐，一般来说，它可以被定义为使目标状态与源状态一致的过程。在本论文中，状态调谐将主要在软件应用程序和基础设施的管理中被考虑，因此，在大多数情况下，目标状态将被视为所需的、用户定义的状态，如 IaC，而实际状态将被视为被管理的平台、基础设施或应用的状态。

2.3.2 持续状态调谐

在很多情况下，调谐过程是按需触发的（例如，由用户手动触发，或因基础设施代码的新变化而自动触发），并执行一次，直至完成。许多 IaC 工具，包括 Terraform，都采用这种方法。

然而，这种方式有一个缺点，那就是只有在调谐过程结束后，两个状态（期望的和实际的）的一致性才能实现。在任何时候，某些事件，例如绕过 IaC 的对基础设施进行的手动变更，都可能导致两个状态之间的不一致和漂移。通过这种一次性的方法，导致的不一致直到下一次触发调谐过程时才会被解决，而这可能要到下一次对 IaC 代码修改时才会发生。

更先进的系统，如 Kubernetes 中的调谐功能，能够通过不断观察调谐循环中的状态并做出反应，持续尝试确保一致性。这种方法在本论文中被称为持续状态调谐。

2.3.2.1 实现技术

一个基本的实现技术是在定时器的提醒下定期执行一个新的调谐过程。虽然朴素，但这个解决方案是合适的，特别是在变化率较低的环境中。然而，环境中的变化越频繁，为了及时对变化做出反应，核对的时间间隔就需要越短。这可能会在非常短的间隔时间内产生负载和性能影响。

一个改进方法是以基于事件的方式执行调谐过程，每次调谐作为对表明发生变化的事件的反应。例如，如果使用图2-1中的 IaC 工作流，那么期望状态

方面的事件可以是版本控制中的一个新变更。此外，调谐系统也需要接收关于基础设施（实际状态）中的资源变化的事件并作出反应。通过对来自双方的事件（期望状态和实际状态）的反应进行调谐，调谐过程只在需要的时候运行，这可以大大改善性能并减少 API 的负载。

基于事件的方法是否可以实现，取决于实际状态背后的平台，它必须支持产生关于其资源变化的事件。Kubernetes 系统原生支持这一点，这是其主要卖点之一。

2.4 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[9]。

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[10]。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[11]。

2.5 Kubernetes

Kubernetes 在很大程度上利用了2.3节中定义的状态调谐模式。本节介绍并进一步深入 Kubernetes，描述它的架构，它对状态调谐的使用，最重要的是它的 API 可扩展性特征，它可以实现并促进自定义用例的持续状态调谐循环，这也是本论文的部分主题。

正如 Kubernetes 的文档中所说，它是一个可移植的、可扩展的、开源的平台，用于管理容器化的工作负载和服务。它还指出，该平台促进了声明式配置和自动化[12]。在高层次上，Kubernetes 作为一个开源的、与供应商无关的平台，以容器的形式托管应用程序和工作负载，这些容器在节点池上被动态地调度。它为网络、存储和其他基础设施层面的问题提供了抽象，并为配置外部资源（如负载均衡器或云中的块存储）提供了集成。

Kubernetes 的诞生源于 Google 内部的容器编排工具 Borg，Google 在 2014 年 6 月[13] 将其开源[14]。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。

2.5.1 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2-3所示为 Kubernetes 架构，由一组相对较小的服务组成，它们主要通过向一个共同的元数据库读写数据进行合作和交流。在一个典型的集群中，有两组节点：建立控制平面集群的主节点和实际运行应用程序的工作节点。

在此基础上，服务可以进一步分成两组：控制平面组件和节点组件。在大多数配置中，控制平面组件只驻留在主节点上，而节点组件是通用的，驻留在所有的节点上[15]。

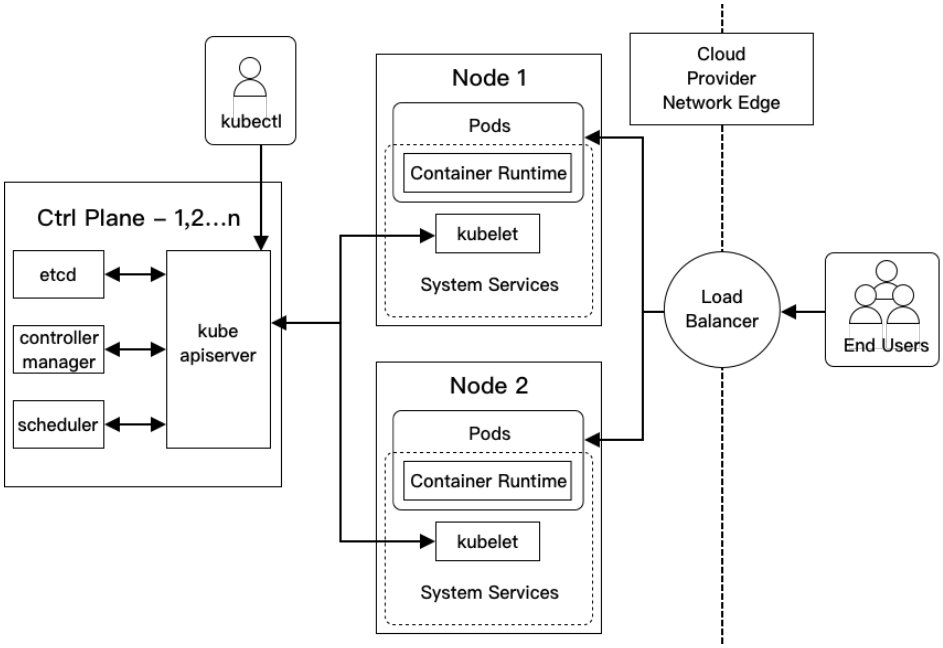

图 2-3: Kubernetes 架构图

2.5.1.1 Kubernetes 控制面

如图2-4所示，Kubernetes 控制面由运行在主节点上的各种组件组成。与本文工作相关的主要有以下几个组件：

etcd（元数据库）

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。它存储了 Kubernetes 的所有配置与状态相关的数据，是系统的核心。

kube-apiserver

kube-apiserver 是整个集群的主要控制模块。etcd 负责可靠地存储所有的元数据，但组件并不直接操作这些数据，而是由 kube-apiserver 提供便利。这个组件作为 etcd 之上的一层，承载着 Kubernetes API，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。“kubectl apply” 是最常用的部署命令。

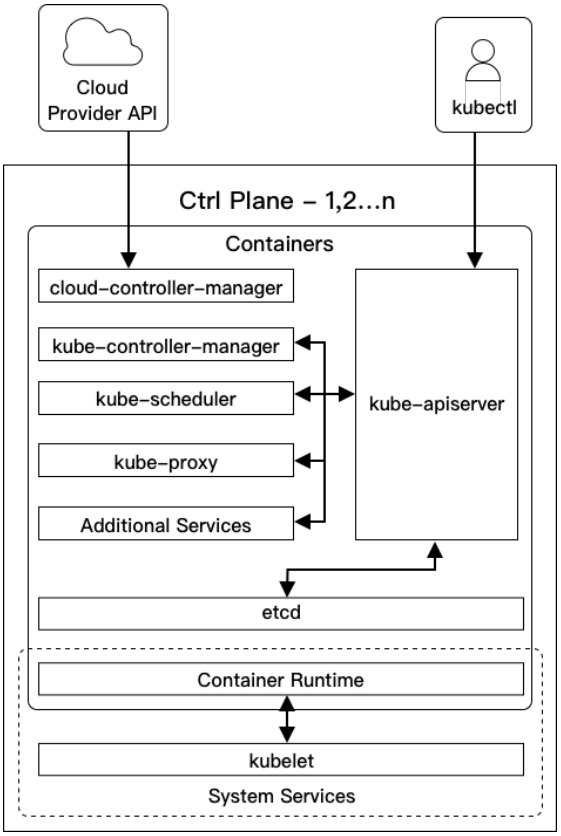

图 2–4: Kubernetes 控制面[1]

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。这里的每个控制器与用户实现的自定义控制器一样都遵循2.5.2.1的控制器模式。

2.5.2 Kubernetes 中的状态调谐

Kubernetes 中的大部分组件和功能都是通过2.3节中描述的持续的、基于事件的状态调谐模式的小型应用实现的。本节阐述这种用法，并通过一个使用 Kubernetes 核心资源类型和组件的例子来说明这一概念。

2.5.2.1 控制器模式

在 Kubernetes 中，状态调谐的使用被包含在被称为控制器的服务中，其名称基于控制理论中的控制循环的概念，是一个调节系统状态的无限循环[16]。

图 2-5: Kubernetes 中的一个控制器

如图2-5所示，一个控制器通过 Kubernetes API 不断监视 Kubernetes 状态中的一些对象的变化。被监视的对象，特别是它们的规格属性，描述了一些期望的状态，控制器将试图在现有环境的相关部分（实际状态）达到这些状态。

换句话说，控制器主持调谐循环，执行如2.3.2节所定义的持续状态调谐。控制器可以对一个事件做出响应，更新 etcd 中的一些资源，或更新一些外部环境（如云提供商的资源），或两者都更新。2.3所定义的期望状态和实际状态的划分，在不同的控制器实现和它们被设计用来解决的问题之间是不同的。

2.5.3 自定义状态调谐

正如本节所讨论的，状态调谐是 Kubernetes 的核心，Kubernetes 及其 API 的许多方面和功能都有助于实现调节循环和控制器。基于调谐的持续控制方法代表了一种通用模式，可以用于实现 Kubernetes 核心功能以外的用例，包括 Kubernetes 特有的自动化或者扩展，也包括完全自定义的应用。这 Kubernetes 可扩展性的核心体现，能够实现具有自定义逻辑的调谐循环。

2.5.3.1 自定义资源定义（Custom Resource Definitions）

为了实现自定义调节，有必要用单独的自定义模式（schema）来定义新的自定义资源。在 Kubernetes 中，这可以通过使用自定义资源定义 CRD 来完成，CRD 是 Kubernetes 中的一种特殊资源类型（CustomResourceDefinition）。

CustomResourceDefinition 类型的资源的结构提供了一些字段，它们被用于指定应添加到 Kubernetes API 中的自定义资源。一旦提交了有效的 CRD，Kubernetes API 将自动扩展新的端点，其 URL 结构取决于一些字段。同样，该

资源也将变得可用，可以通过 kubectl 命令行客户端查询和操作。该资源将自动支持所有 Kubernetes API 动作。核心字段的意义如下：

API Group

API Group 是一个需要为新的自定义资源指定的属性。API Group 的概念被用于大多数 Kubernetes 资源（除了像 Pod 这样的核心资源），将相关的资源类型组合在一起，并创建一个 Kubernetes API 的逻辑 “分区”。该组成为 API 端点的 URL 的一部分，也是自定义资源元数据的一部分，也可用于权限控制。使用域名来命名 API 组是很常见的，以表明所有权或作者。

由于 CustomResourceDefintion 本身也是一种 Kubernetes 资源，如 YAML 文件 4.1第 1 行所示，它所属的 API Group 是 apiextensions.k8s.io

Versions

CRD 的规范还需要为新资源指定至少一个版本。版本指定了新资源结构的模式，使用 OpenAPI6 的模式规范格式。每个版本都可以被启用或禁用，而且其中一个版本必须被配置为存储版本。存储版本代表实际将被存储在 etcd 中的结构，而其他版本将简单地被转换为存储版本。

Names

要声明一个新的 Kubernetes API 资源，需要指定其名称的几种形式。复数名称（YAML 文件 4.1第 12 行的 universalcontrollers）将被用于 Kubernetes API 端点的资源的 URL 结构中，即/apis/universalcontroller.njuics.cn/v1alpha1/universalcontrollers。单数名称（第 10 行）通常用于 kubectl CLI 命令，例如，kubectl get universal-controller。也可以为其定义更短的别名，如 YAML 文件 4.1的第 13-15 行所示。最后，必须指定的最后一个名称是 kind，它是资源类型的名称，在配置清单中使用。

Scope

CRD 还必须指定新资源的范围。一般来说，在 Kubernetes 中，一个特定的资源类型可以是命名空间范围的，即每个对象必须属于某个命名空间，或者是集群范围的，即每个对象是全局的。在 CRD 中，这是用范围参数指定的，如 YAML 文件 4.1的第 17 行。

2.5.3.2 自定义控制器

一旦新的 CRD 被注册，各自的自定义资源对象及其规格就可以被应用到 Kubernetes。虽然这允许声明和存储所需的状态，但需要有一个自定义控制器，以便将规范应用到真实的状态或环境中。自定义控制器需要使用 watch API 去监听这个新资源对象的相关事件，并对事件作出响应行为。

自定义控制器不是 Kubernetes 的一部分，需要单独安装。由于所有的逻辑和依赖都包含在控制器的代码中，安装可以通过简单地在容器中部署控制器来完成，使用与其他应用程序相同的方法，如添加一个新的 Pod 或一个 Deployment（一组副本 Pod）对象。

2.5.4 Operator 模式

2.5.3节描述的方法通常被称为 Operator 模式。这个术语基本上是指将控制器模式（2.5.2.1节）应用于一些定制的、特定领域的用例。遵循这种模式的服务被称为 Operator，并与一个 CRD 配对。

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”[17]。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，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，负责管理的 Kubernetes Operator 会对对象发出指令，使其最终达到期望状态。下面介绍一些开发者实现 Operators 所使用的主要工具。

2.5.4.1 主要的开发工具

client-go

client-go 是最成熟的 Kubernetes 客户端库，它和 Kubernetes 自身一样都由 Go 语言编写而成，是 Kubernetes 官方第一个提供的客户端库，而且正在被 Kubernetes 本身的组件内部使用，这也意味着它是经过良好测试和可靠的[18]。

该库封装了几个抽象和较小的包，便于实现 Kubernetes 的集成和与 Kubernetes API 的通信。有三个包与基本的 Kubernetes API 通信有关[18]：

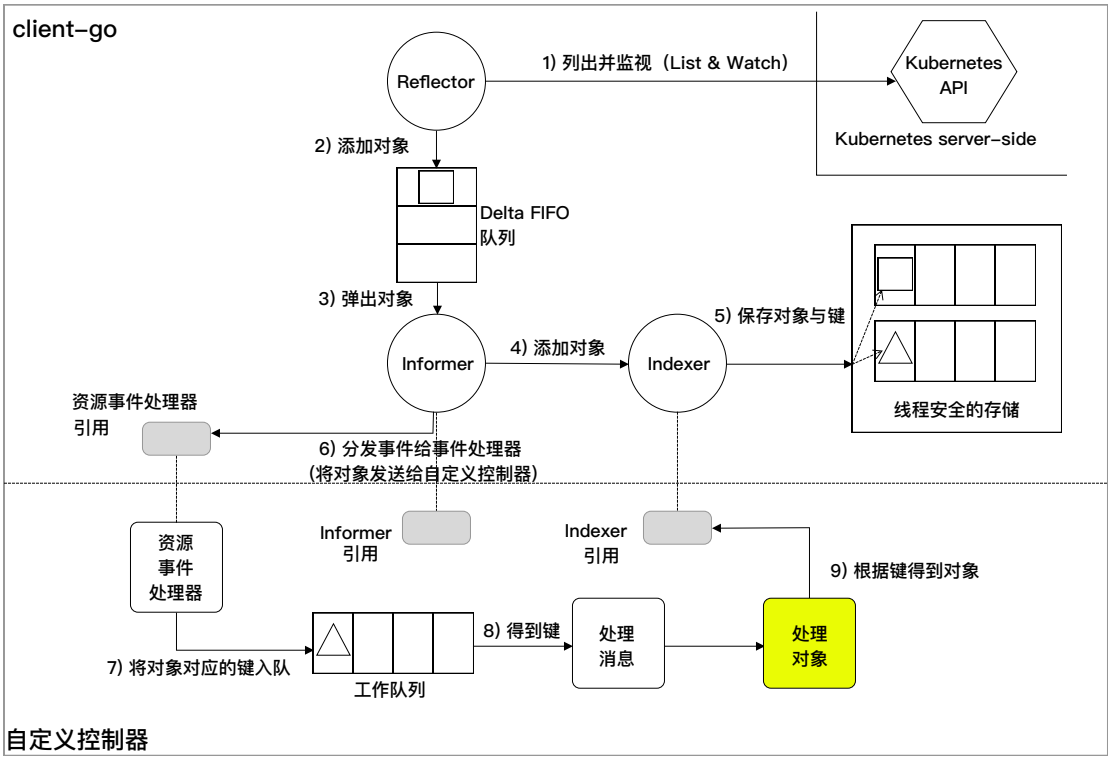

图 2-6: 控制器工作方式

- kubernetes 包，提供静态（和静态类型）的客户端，可用于对 Kubernetes API 执行涉及 Kubernetes 原生资源类型的操作。
- dynamic 包，它提供了一个动态客户端，能够对任何资源类型（原生和自定义）进行通用操作。
- transport 包，在与 Kubernetes 通信时帮助处理低级别的传输细节，如使用有效的认证建立连接等。

此外，由于 client-go 在整个 Kubernetes 代码库中被使用，包括复杂的场景，它还附带了额外的工具、实用程序、对象和抽象，以简化 Kubernetes 集成。

鉴于本文的主题与状态调谐和自定义 Kubernetes 控制器有关，客户端最引人注目的功能是 Informer 模式的实现[19]。

Informer 是对 Kubernetes API 的实时观察功能的抽象，API 可以将集群中任何对象的任何变化事件通知消费者。它们提供了一个接口，允许开发者为特定的 Kubernetes 资源类型有效地建立所述的变化流连接[19]。这代表了实现自定义控制器的关键功能，它需要不断监视和应对资源相关变化的。

图2-6展示了使用 clieng-go 编写的控制器的工作方式。首先介绍一下

client-go 提供给开发者的组件：

1. **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。
2. **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。
3. **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以下组件：

1. **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。
2. **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。
3. **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。

controller-runtime

controller-runtime 建立在 client-go 库之上，用构建控制器的相关概念和 API 来扩展它，其中包括[20] 读写 Kubernetes 对象的高级客户端、用于高效获取 Kubernetes 对象的缓存、用于共享依赖关系和启动控制器的管理器（Manager）、核心抽象控制器、用于扩展 Kubernetes API 的对象准入流程的 Webhook、由事件触发执行的调谐器、封装 Kubernetes 事件流的源。

2.6 小结

本章主要介绍了声明式的通用 Kubernetes Operator 设计到的相关技术和场景。首先介绍了动态基础设施及其标准。之后解释 IaC 的概念，并描述了相关的实践和有点，介绍了支持这些实践的不同类型的工具，并在 IaC 的背景下讨论了指令式和声明式编程范式。接着介绍 IaC 的核心机制之一状态调谐，最后深入研究了 Kubernetes 平台的细节，描述了它如何拥抱声明式并且使用持续调谐循环。

第三章　声明式的通用调谐技术

本文所陈述的声明式的通用 Kubernetes Operator，即 UniversalController，实现了声明式的通用调谐技术，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，而实现这一点的核心就是声明式调谐技术。

3.1　现有的 Kubernetes Operators 实现方式

Kubernetes 的 Operator 可以简单解释为自定义资源与自定义控制器的组合。自定义资源通过 Kubernetes 的 CustomResourceDefinition 机制可以很容易地生成，而自定义控制器需要用户自行编写。

编写自定义控制器是编写代码来管理一个应用程序的生命周期的过程。它需要付出极大的努力，但对各种特殊的要求有最大的自由。这种方法最好用于需要非常特殊功能的应用程序的操作。

开发一个自定义的控制器需要很多对 Kubernetes 的深入知识，而它们通常对大多数应用程序的生命周期管理是不需要的。除了应用程序生命周期的具体要求外，它还会给开发者带来管理负担，包括测试、升级、改变 Operator 的存储数据和改变 API[21]。

3.1.1　Kubernetes Clients

在开发控制器时，最自由的做法是使用现有的 Kubernetes 客户端，可以是 Go、Java、JS/Typescript 或其他语言的客户端。这些客户端提供了对 Kubernetes APIs 的直接和底层的访问，没有任何包装或附加层。

这允许使用任何语言，并且不将任何观点强加给开发者。但是，它要求开发者解决很多已知的问题，并且没有为 Kubernetes 领域的常见问题提供指导[21]。

3.1.1.1 client-go

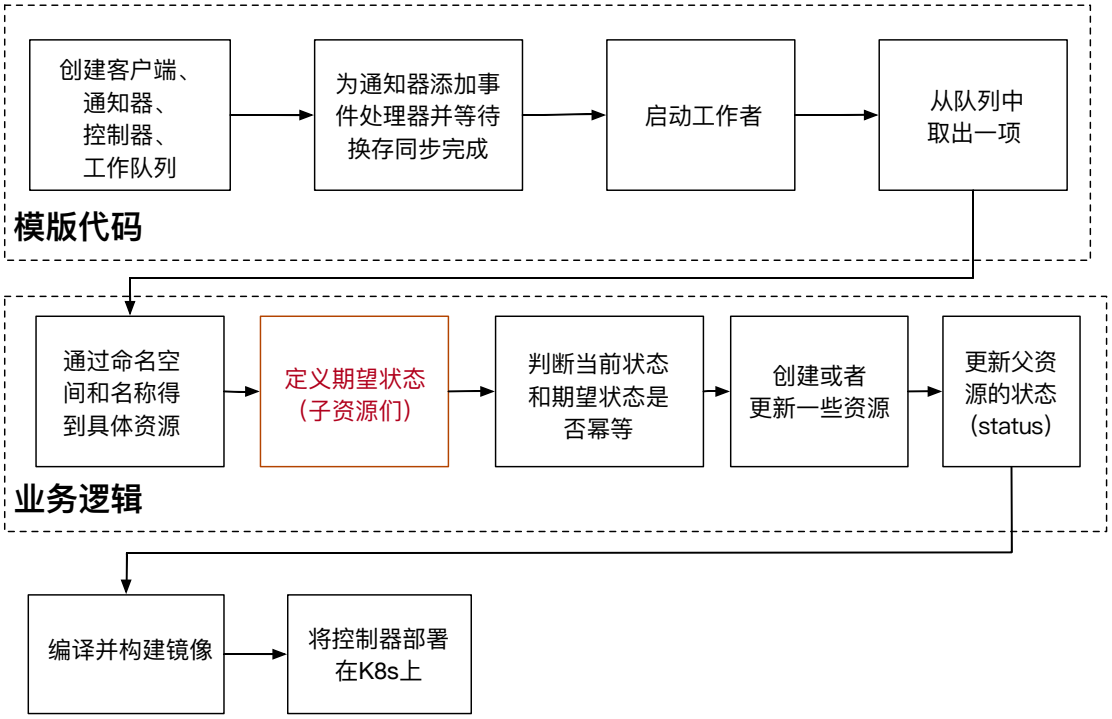

图 3-1: Operator 编写流程

client-go 是最常用也最成熟的客户端库，图 2-6描述了控制器的工作方式，图 3-1是使用 client-go 开发一个自定义控制器的基本流程，除了最后用于部署控制器的两步以外，各个流程都与图 2-6中的一个组件或者一个步骤对应。client-go 相关代码以模版代码居多，在不同的控制器中，创建客户端、通知器、工作队列的代码都是高度相像的，只是需要处理的资源类型不同。每个控制器真正核心的部分都是“定义期望的状态（子资源们）”，即给出期望状态（state）。

3.1.2 Kubernetes Controller Runtime

正如上一节所讨论的，client-go 库提供了许多抽象，可以简化 Kubernetes 控制器和状态调谐的实现。尽管如此，该库是为了成为一个通用的客户端而设计的，它并没有专门去解决编写控制器的很多问题。

其他项目，如 Operator SDK 和 Kubebuilder，提供了更高层次的抽象。它们专门针对 Kubernetes API 扩展的开发者，并考虑到自定义控制器的设计。它们

都建立在一个共同的核心代码库之上，即 controller-runtime[20]。它是一组库，共同代表了用自定义调谐逻辑扩展 Kubernetes 的通用模型[22][20]。

借助 controller-runtime 以及代码生成工具，我们可以省去很多模版代码的编写，但是我们依然必须用 Go 语言来开发 Operator 项目，依然需要与 Kubernetes 的 API 打交道，需要自行处理更新策略，而且也不能规避所有的模版代码，判断实际状态和期望状态是否幂等、对资源进行创建或者更新依然是一套惯例代码样式。

3.1.3 问题分析

在开发 Kubernetes Operator 时，开发者最关心也是最核心的部分就是调谐逻辑，但是为了实现一个完整的 Operator，开发者不得不把大量的精力放在编写一些模版代码或者使用代码生成工具上，而且 Kubernetes Operator 的成熟开发工具都是用 Go 编写的，也是为 Go 项目服务的，对使用其他编程语言的开发者极不友好。而为了使用这些工具，用户也必须去学习 Kubernetes API 相关的深度知识，进而拉高了门槛。

本文的目的是简化 Kubernetes Operators 的开发、部署和管理，而通过声明式的通用调谐技术就能很好的做到这一点，接下来开始介绍。

3.2 声明式的通用调谐技术

声明式的通用调谐技术的初衷十分简单，既然开发者实际只关心调谐逻辑，而其他与 Kubernetes 相关的代码基本都是模版代码或者有惯例可循，那么完全可以将这部分代码代码单独抽取出来，提供一个接口供用户实现，其他的代码都由系统或者框架代劳。基于这个想法实现的就是 UniversalController，一个声明式的通用 Kubernetes Operator，它实现了通用的资源监视、当前状态与期望状态对比、资源更新等功能，提供声明式的接口，帮助用户通过声明式编程实现 Operator，减少了重复工作。

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。UC CRD 是对一个自定义控制器的抽象描述。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，UC Controller 会通过调谐循环维护它。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。借助 UniversalController，一个 Operator 的开发会简化成如图 3-2所示。图 3-2中的 Kubeless 是一个 serverless 工具，主要用于将一个函数部署成一个网络服务。

图 3-2: 借助 UC 实现一个 Operator

3.2.1 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。用户只需要填写好各项字段就能声明式地创建想要的控制器。通过 UC CRD 实现了以下功能：

1. **声明式的监视（watch）**：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。
2. **声明式的更新策略**：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。
3. **声明式的调谐**：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。用户在编码过程中只需要描述期望的状态即可，而不需要给出具体操作以达到期望状态，这个过程本质上也是声明式的。

3.2.2 声明式的监视

借助 UC，开发者不再需要为想监视的每一种类型的资源编写模板代码，而只要简单地列出这些资源的声明。YAML 段 3.1是 5 .1节的 YAML 文件 5.1的第 7 到 12 行，说明该自定义控制器的父资源是 Foo（foos 是 Foo 在 K8s 中的复数表达），子资源是 Deployment（deployments 是 Deployment 在 K8s 中的复数表达）。UC 会负责建立监视流（watch stream），这些监视流被所有使用 UC 创建的控制器共享。开发者可以在 UC 之上创建任意多的自定义控制器来观察 Pod，而 kube-apiserver 只需要发送一个 Pod 监视流。UC 就像一个解复用器，确定哪些控制器会关注流中的特定事件，并根据需要触发它们的事件处理器（一个函数）。

Listing 3.1: 声明式的监视

```
1 parentResource:
2   apiVersion: njuics .cn/v1alpha1
3   resource: foos
4 childResources:
5 - apiVersion: apps/v1
6   resource: deployments
```

而如果用 client-go 来编写，需要先创建通知器工厂，再用工厂创建通知器，最后添加事件处理器，对资源的增加、更新、删除做出反应，如代码段 3.2所示。代码段 3.2的第 5 到 10 行的意思是，对于 Foo 资源，它的增加、更新和删除触发的事件处理器的工作都是把相关 Foo 资源的键加入工作队列等待处理，这是对父资源处理的一般惯例。对于 Deployment 资源，它的增加、更新和删除触发的事件处理器都调用 handleObject 方法。handleObject 的简化版是代码段 3.2的第 22 到 27 行，意思是根据 Deployment 的 OwnerReference 信息得到它的父资源 Foo 的命名空间和名称以定位，将这个父资源的键加入工作队列。这是对子资源处理的一般惯例。代码段 3.2是典型的模版代码，不同的控制器都遵循这样的惯例，只是资源类型不同而已。

Listing 3.2: sample-controller 中监视 Foo 和 Deployment 的代码段

```
1 kubeInformerFactory := kubeinformers.NewSharedInformerFactory(
     kubeClient, time.Second*30)
2 exampleInformerFactory := informers.NewSharedInformerFactory(
     exampleClient, time.Second*30)
```

```
3  fooInformer := exampleInformerFactory.Samplecontroller().V1alpha1
       ().Foos()
4  deploymentInformer := kubeInformerFactory.Apps().V1().Deployments
       ()
5  fooInformer.Informer().AddEventHandler(cache.
       ResourceEventHandlerFuncs{
6      AddFunc: controller.enqueueFoo,
7      UpdateFunc: func(old, new interface{}) {
8          controller.enqueueFoo(new)
9      },
10     DeleteFunc: controller.enqueueFoo,
11 })
12 deploymentInformer.Informer().AddEventHandler(cache.
       ResourceEventHandlerFuncs{
13     AddFunc: controller.handleObject,
14     UpdateFunc: func(old, new interface{}) {
15         newDepl := new.(*appsv1.Deployment)
16         oldDepl := old.(*appsv1.Deployment)
17         if newDepl.ResourceVersion == oldDepl.ResourceVersion {
               return }
18         controller.handleObject(new)
19     },
20     DeleteFunc: controller.handleObject,
21 })
22 func (c *Controller) handleObject(obj interface{}) {
23     if ownerRef := metav1.GetControllerOf(object); ownerRef != nil
           {
24         if ownerRef.Kind != "Foo" { return }
25         foo, err := c.foosLister.Foos(object.GetNamespace()).Get(
               ownerRef.Name)
26         if err != nil { return }
27         c.enqueueFoo(foo)
28         return
29     }
30 }
```

代码段3.3借助 controller-runtime 库设置监视（watch）。借助 controller-

runtime 提供的高级接口，相比代码段 3.2要精简的多。但依然要比 YAML 文件片段 3.1复杂，还要考虑到 Go 语言和 controller-runtime 接口的学习成本，所以使用依旧比较困难。

Listing 3.3: controller-runtime 版 sample-controller 中监视 Foo 和 Deployment 的代码段

```
1  c.Watch(&source.Kind{Type: &samplev1alpha1.Foo{}}, &handler.
       EnqueueRequestForObject{})
2  subresources := []runtime.Object{
3      &appsv1.Deployment{},
4  }
5  for _, subresource := range subresources {
6      c.Watch(&source.Kind{Type: subresource}, &handler.
           EnqueueRequestForOwner{
7          IsController: true,
8          OwnerType: &samplev1alpha1.Foo{},
9      })
10 }
```

3.2.3 声明式的通用调谐器

图 4-1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：

1. 根据集群的当前状态（state）得到资源的状态（status）。
2. 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。

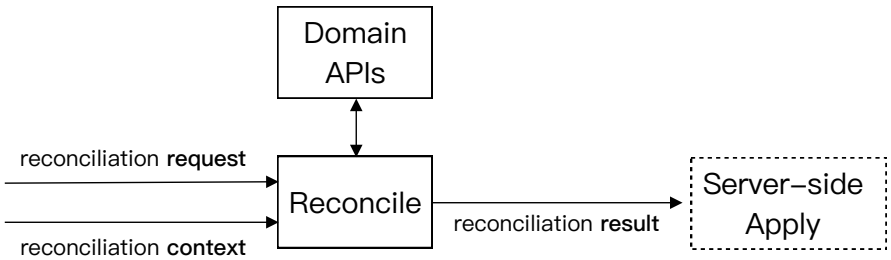

图 3-3: 调谐器

图 3-3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。

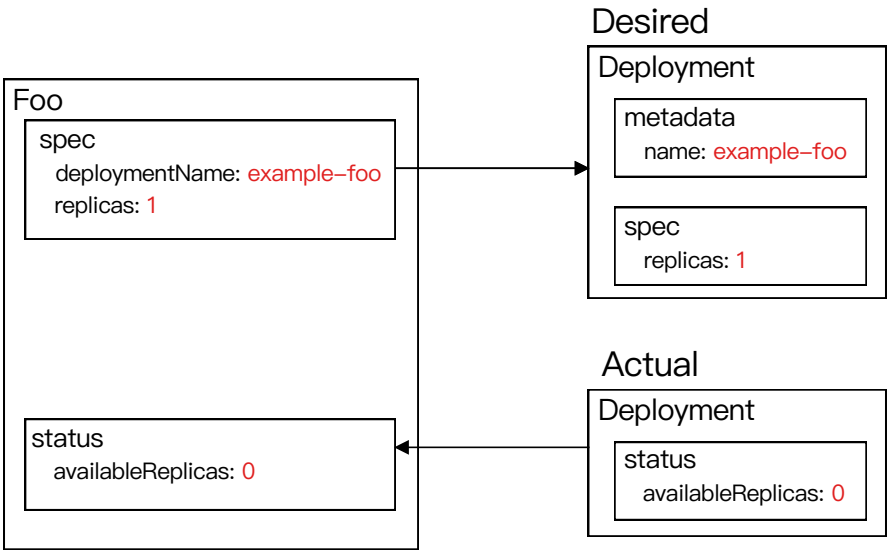

图 3-4: sample-controller 的调谐过程

举例来说，5.1节的 sample-controller 负责管理的自定义资源是 Foo，它的 spec 中只有两个字段 deploymentName 和 replicas，代表期望生成的 Deployment 资源的名称和指定的副本数量，它的 status 中只有一个字段 availableReplicas，代表集群中实际可用的副本数量。图 3-4展示了 sample-controller 的一次调谐过程。根据这个 Foo 的 deploymentName 和 replicas 会去生成一个名称为 example-foo，副本数量为 1 的 Deployment，这就是第二个翻译过程，对应代码 5.2的第 17 到 41 行的 desiredDeployment 函数。而根据当前集群中实际的 Deployment 的 status 中的 availableReplicas 字段可以得知可用的副本数量此刻为 0，所以 Foo 的 status 的 availableReplicas 字段的值也要设置为 0，这就是第一个翻译过程，对应代码 5.2的第 10 到 13 行。

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 kube-apiserver 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，自定义资源和原生资源都根据一套“kubectl 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)}。例如，5.1节的代码段5.2中的 reconcile 就是这样一个函数，最后只要返回调谐结果即可，不需要做任何增删改查之类的指令式操作。而使用 client-go 或者 controller-runtime 实现调谐逻辑时，都需要开发者调用接口对资源进行增删改查。代码段3.4是用 client-go 编写时增删改查资源相关的代码。第 1 行获取本次调谐处理的 Foo 资源对象；第 2 行获取已经存在的子资源 Deployment；第 3 到 5 行表示如果子资源还不存在，就创建它；第 6 到 8 行表示如果已存在的 Deployment 的 replicas 与 Foo 中的不一致，就更新它；第 9 行更新 Foo 的 status。

Listing 3.4: 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{})
```

3.2.4 服务端应用（apply）

3.2.5 声明式的更新策略

一般的 Operator 在比较期望状态和实际状态后，会对资源进行更新，进行增删改查的操作。而不同的资源适合不同的更新策略，例如 Pod 一般会用重新创建，因为当一个 Pod 已经在 Kubernetes 中存在，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，直接更新原资源即可。

在借助 UC 实现的自定义控制器中，UC Controller 会代为执行更新相关操作，为了更灵活的进行更新，具体资源具体分析，内置了很多更新策略，通过声明式接口供用户使用。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，内置了相关的更新策略，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。例如，5.3节中的 CatSet 用例是在基于 UC 重写的 StatefulSet，为其添加对滚动更新的支持，只需要 YAML 文件5.5中的第 16 到 21 行，也就是 YAML 片段3.5中的第 4 到 9 行，表示对 Pod 更新采取滚动重写创建的策略。当 Pods 需要被更新时，每次按照指定的粒度重新创建一批，这一批的 Pods 的 status 中 condition 字段都包含片段3.6时，也就是都就绪时，才会去更新下一批。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[23]。

为 StatefulSet 添加滚动更新支持为 Kubernetes 引入了 ControllerRevision 的概念，用于对资源进行版本控制，保存滚动更新的中间信息，确保滚动更新被中断或者遭遇故障（例如控制器被删除重建）后可以恢复。UC 也使用 ControllerRevision 对资源进行版本控制，以支持滚动更新，但存储的中间信息结构与 StatefulSet 有些差异，具体的会在4.5.2节介绍。

Listing 3.5: 添加滚动更新

```
1     childResources:
2     - apiVersion: v1
3       resource: pods
4   +   updateStrategy:
```

```
+      method: RollingRecreate
+      statusChecks:
+        conditions:
+        - type: Ready
+          status: "True"
```

Listing 3.6: 状态 Ready 为 True

```
  - lastProbeTime: null
    lastTransitionTime: "2021-04-15T06:44:53Z"
    status: "True"
    type: Ready
```

3.3 小结

本章首先对一个 Opeator 现有开发流程进行了介绍，然后在此基础之上总结其中存在的问题，之后提出声明式的调谐技术，致力于解决这些问题。

第四章　声明式的通用 Kubernetes Operator 的设计与实现

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 的关键之一。

4.1　总体架构

图 4-1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。

4.2　自定义资源

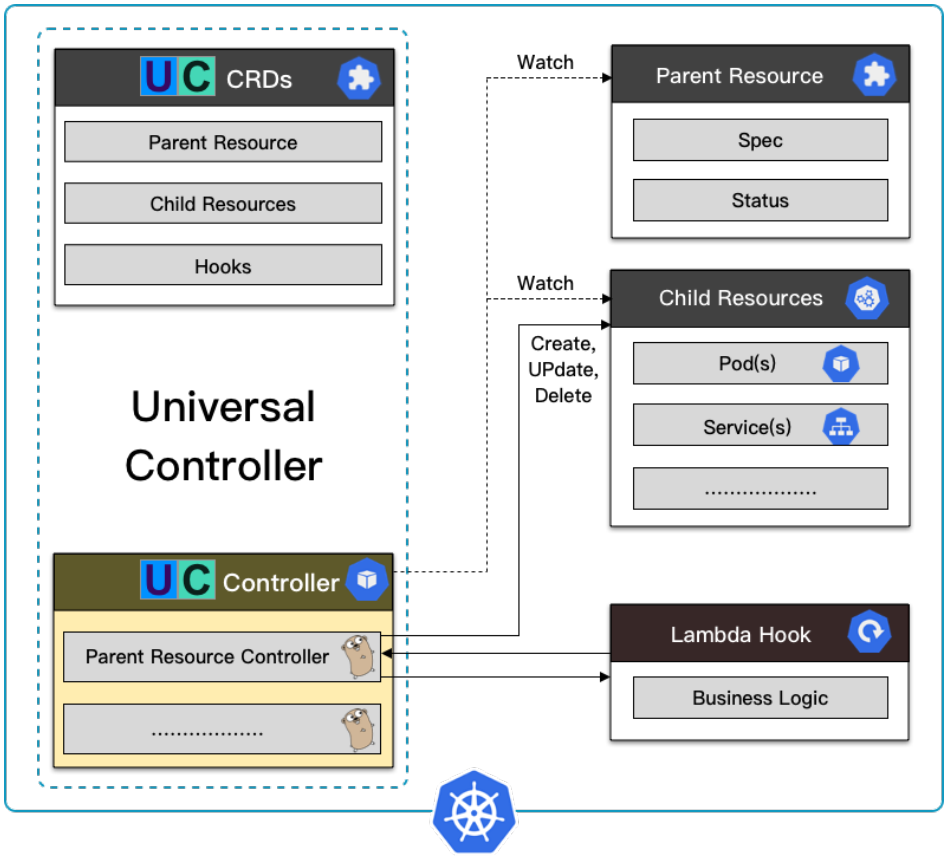

图 4-1: UniversalController 架构

Listing 4.1: UC CRD

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    "api-approved.kubernetes.io": "unapproved, request not yet
        submitted"
  name: "universalcontrollers.universalcontroller.njuics.cn"
spec:
  group: "universalcontroller.njuics.cn"
  names:
    kind: UniversalController
    listKind: UniversalControllerList
    plural: universalcontrollers
    shortNames:
    - 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” 有五个字段：parentResource、childResources、resyncPeriodSeconds、generateSelector 和 hooks。parentResource 的类型是 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 是是一组 ResourceRule，用于指定会被控制器生成的子资源类型。resyncPeriodSeconds 被用于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 是 bool 型的，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 是一组定义了控制器行为的 lambda hook。

4.3 动态类型高级操作接口实现

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。

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)
```

```
12 func (cs *Clientset) Resource(apiVersion, resource string) (*
       ResourceClient, error) {
13     // Look up the requested resource in discovery.
14     apiResource := cs.resources.Get(apiVersion, resource)
15     if apiResource == nil {
16         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

```
// sharedResourceInformer is the actual, single informer that's shared by
// multiple ResourceInformer instances.
type sharedResourceInformer struct {
   informer cache.SharedIndexInformer
   lister dynamiclister.Lister
   defaultResyncPeriod time.Duration
   eventHandlers *sharedEventHandler
   close func()
}
func newSharedResourceInformer(client *dynamicclientset.ResourceClient, defaultResyncPeriod time.Duration, close func()) *sharedResourceInformer {
   informer := cache.NewSharedIndexInformer(
      &cache.ListWatch{
         ListFunc: func(opts metav1.ListOptions) (runtime.Object, error) {
            return client.List(opts)
         },
         WatchFunc: client.Watch,
      },
      &unstructured.Unstructured{},
      defaultResyncPeriod,
      cache.Indexers{
         cache.NamespaceIndex: cache.MetaNamespaceIndexFunc,
      },
   )
   sri := &sharedResourceInformer{
      close:          close,
      informer:       informer,
      defaultResyncPeriod: defaultResyncPeriod,
      lister: dynamiclister.New(informer.GetIndexer(), client.GroupVersionResource()),
   }
```

```
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。

4.4 控制器实现

图4-1中的 UC Controller 和 Parent Resource Controller 都是自定义控制器。图2-6展现了一个自定义控制器的工作方式。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。

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

4.4.1 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。

4.4.2 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 }
```

4.5 声明式的更新策略

4.5.1 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）的子资源，并根据期望状态（state）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）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。

4.5.2 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

```
apiVersion: universalcontroller.njuics.cn/v1alpha1
kind: ControllerRevision
metadata:
  name: catsets.universalcontroller.njuics.cn-5463
    ba99b804a121d35d14a5ab74546d1e8ba953
  labels:
    app: nginx
    component: backend
    universalcontroller.njuics.cn/apiGroup: universalcontroller.njuics.cn
    universalcontroller.njuics.cn/resource: catsets
parentPatch:
  spec:
    template:
      [...]
children:
- apiGroup: ""
  kind: Pod
  names:
  - nginx-backend-0
  - nginx-backend-1
  - 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。

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

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。

4.6 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。

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

Hooks

在 UC CRD 的 spec 中，hooks 字段有以下三个子字段：sync、finalize 和 customize。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。它们都对应了一种 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 “server-side 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，它的字段包括 parent、children、related 和 finalizing。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 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 对象只有当 Customize 钩子存在时，会存储相关资源，否则为空。finalizing 是布尔型的，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

同步钩子的响应体的字段包括 status、children 和 resyncAfterSeconds。status 是一个 JSON 对象，将完全取代父资源中的状态字段。children 是一组 JSON 对象组成的列表，代表所有期望存在的子资源。resyncAfterSeconds 是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，如果调谐逻辑拒绝在响应体中返回，它会被 UniversalController 在收到响应后删除。但是，调谐逻辑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 的字段包括 apiVersion、resource、labelSelector、namespace 和 name。apiVersion 是资源的 apiVersion，例如 apps/v1、v1、batch/v1 等。resource 是资源的小写名称，例如 deployments, replicasets, statefulsets。labelSelector 是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 是选填项，指资源所在的命名空间。name 也是选填项，代表资源名称列表。

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。

4.7 小结

本章详细介绍了各个组件或功能在 UniversalController 中是设计与实现的，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 的配置文件

```
1  apiVersion: universalcontroller .njuics .cn/v1alpha1
2  kind: UniversalController
3  metadata:
4    name: sample-controller
5  spec:
6    generateSelector: true
7    parentResource:
8      apiVersion: njuics .cn/v1alpha1
9      resource: foos
10   childResources:
11   - apiVersion: apps/v1
12     resource: deployments
13     updateStrategy:
14       method: InPlace
15   hooks:
```

```
16    sync:
17      webhook:
18        url: "http://sample-controller.universalcontroller:8080"
```

YAML 文件5.1是 sample-controller 的定义文件，7-9 行指定了它需要处理的父资源类型，10-12 行指定了子资源类型。18 行是用户的调谐逻辑的入口。

Listing 5.2: sample-controller 的实现代码

```
1  module.exports = {
2    handler: (event, context) => {
3      let observed = event['data'];
4      return reconcile(observed.parent, observed.children);
5    }
6  };
7  var reconcile = function (foo, children) {
8      let desiredChildren = [];
9      let currentStatus = {};
10     let allDeploys = children['Deployment.apps/v1'];
11     let fooDeploy = allDeploys ? allDeploys[foo.spec.
           deploymentName] : null;
12     let replicas = fooDeploy ? fooDeploy.status.availableReplicas
           : 0;
13     currentStatus = {availableReplicas: replicas};// Set the
           status of Foo
14     desiredChildren = [desiredDeployment(foo)];
15     return {status: currentStatus, children: desiredChildren};
16 }
17 var desiredDeployment = function (foo) {
18   let lbls = {app: "sample"};
19   let deploy = {
20     apiVersion: "apps/v1",
21     kind: "Deployment",
22     metadata: {
23       name: foo.spec.deploymentName,
24       namespace: foo.metadata.namespace,
25     },
26     spec: {
```

```
    replicas: foo.spec.replicas,
    selector: {matchLabels: lbls},
    template: {
      metadata: {labels: lbls},
      spec: {
        containers: [{
           name: "nginx",
           image: "nginx:stable"
         }]
      }
    }
  };
  return deploy;
};
```

JavaScript 代码 5.2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。最后使用 “kubectl apply” 命令提交 YAML 文件 5.1完成控制器的注册。

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 5.3: 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 5.3是 tensorflowjob-controller 的声明式定义。

Listing 5.4: tensorflowjob-controller 的实现代码

```
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) {
      desiredChildren.push(newSVC(tfjob, rtype, index));
    }
    let podSlice = podSlices[index];
    if (podSlice.length === 1) {
      let pod = podSLice[0];
      let exitCode = getContainerExitCode(pod);
      if (spec.restartPolicy === 'ExitCode') {
        if (pod.status.phase === 'Failed' && isRetryableExitCode(
           exitCode)) {
          updateJobConditions(status, 'Restarting', 'TFJobRestarting
             ',
             `TFJob ${tfjob.metadata.name} is restarting because ${
                rtype} replica(s) failed.`);
          continue;
        }
      }
      if (pod.status.phase === 'Running') {
        status.replicaStatuses[rtype].active++;
      } else if (pod.status.phase === 'Succeeded') {
        status.replicaStatuses[rtype].succeeded++;
      } else if (pod.status.phase === 'Failed') {
        status.replicaStatuses[rtype].failed++;
      }
```

```
34      }
35      if (index < numReplicas) {
36        let masterRole = isMasterRole(replicas, rtype, index);
37        desiredChildren.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 5.4。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。

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 5.5: catset-controller 的配置文件

```
1  ---
2  apiVersion: universalcontroller.njuics.cn/v1alpha1
3  kind: UniversalController
```

```
4  metadata:
5    name: catset- controller
6  spec:
7    parentResource:
8      apiVersion: mlhub.njuics .cn/v1alpha1
9      resource: catsets
10     revisionHistory:
11       fieldPaths:
12         - spec.template
13   childResources:
14     - apiVersion: v1
15       resource: pods
16       updateStrategy:
17         method: RollingRecreate
18         statusChecks:
19           conditions:
20             - type: Ready
21               status: "True"
22     - apiVersion: v1
23       resource: persistentvolumeclaims
24   hooks:
25     sync:
26       webhook:
27         url: "http://catset-controller.universalcontroller:8080"
28     finalize :
29       webhook:
30         url: "http://catset-controller.universalcontroller:8080"
```

Listing 5.6: CatSet 资源示例

```
1  apiVersion: mlhub.njuics .cn/v1alpha1
2  kind: CatSet
3  metadata:
4    name: nginx-backend
5  spec:
6    serviceName: nginx-backend
7    replicas: 3
8    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5.5是 catset-controller 的声明式定义。YAML 文件5.6是一个 CatSet 资源的示例，CatSet 的 spec 结构与 StatefulSet 的 spec 结构完全一样。

Listing 5.7: 生成 Pod 和 PVC

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

```
  let desiredChildren = [];
  let currentStatus = {};
  let finalized = false;
  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.
    finalized = (Object.keys(observedPods).length === 0);
  }
  for (var ready = 0; ready < catset.spec.replicas &&
      isRunningAndReady(observedPods[ready]); ready++) ;
  currentStatus = {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) {
```

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

```
64   },
65 };
```

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。

5.4 用例对比与分析

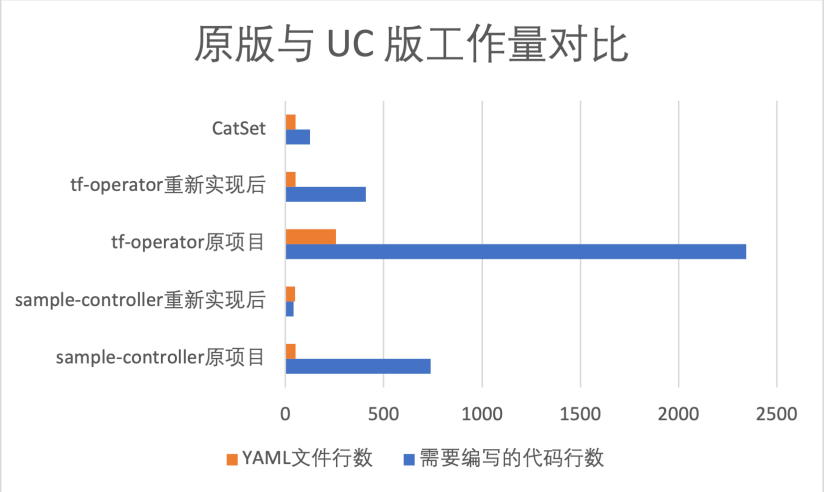

图 5-1: 原版与 UC 版代码行数比较

图 5-1直观地展示了直接用 client-go 实现 Operator 与借助 UC 实现在工作量上的巨大差距。

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。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。

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

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

现复杂 Operator 时依然保持工作量相对较小的能力。

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 5.5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[23]。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。

5.5 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。

5.5.1 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。

5.5.2 对比方法

我设计了八种场景，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。我还实现了一种 donothing-operator，它的 CRD 为 DoNothing-< 随机后缀 >，它的控制器会订阅 Pod 和 PVC，但是什么都不干。实现它的目的是为了方便实验。

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。
- **场景 7**：安装两个 donothing-operator 和原版的 tf-operator。
- **场景 8**：先安装 UniversalController，再安装重新实现的 tf-operator 以及 catset-operator，最后安装用 UC 重新实现的 donothing-operator。

一个 Opeator 在安装之后，首先会与 kube-apiserver 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。

5.5.3 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 的负载很接近。tf-operator 和 catset-operator 都需要订阅 Pod 资源，但是当它们都部署在

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
7	37.84	15839.75
8	23.13	13708.53

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。场景 8 相对场景 6 的资源增量，以及场景 7 相对场景 5 的资源增量也反映了这一点。

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。

5.6 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，尤其是声明式调谐接口，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” 命令部署相近的体验，只是需要改用 JSON 编写一些配置文件，并且可以使用任意自己熟悉或者喜欢的编程语言来实现。如果用户已经很熟悉用 “kubectl apply” 命令去使用 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。

总而言之，本文工作的主要贡献包括：

1. 针对 Operator 开发困难的问题，提出一种声明式的通用调谐技术，简化需要编写的代码，大量减少 Operator 开发者的工作量，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，即描述期望状态上。
2. 实现了声明式的通用 Kubernetes Operator，UniverslController。该工具具有声明式的资源监视（watch），声明式的调谐、声明式的更新策略和语言无关的特性。用户不需要编写任何与 Kubernetes 交互的代码，只需要在 YAML 文件中描述需要监听的资源、使用的更新策略以及在调谐代码段中描述期望的状态即可。
3. 基于 UniversalController 重新实现了一些现有的 Operator，证明了 Universal-Controller 可以极大的缩减开发工作量，并且适用于大部分场景的开发。同时性能测试验证了它还能在多自定义控制器部署的环境中减少内存消耗和 kube-apiserver 的负载。

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。

致　　谢

硕士漫漫三年这么快就过去了，在此期间我相识很多为我提供很多帮助或欢乐的人，由衷地感谢他们。

首先，也是最主要感谢的是我的指导老师，曹春老师。在我的硕士阶段都及时适当的提点我，让我思路贯通，他的细心指导是我顺利完成研究的最大助力。

我还要感谢我的朋友们，谢谢大家三年来给我的关心、信任和帮助，谢谢你们陪我走过人生一段美好时光。

最后，深深感谢我的父母和亲人。这些年，您们无私而无微不至的关心和鼓励，让我从不孤单。

在此，我衷心感谢所有帮助我的人，没有你们我不可能完成这项工作，没有你们我的三年不会如此充实，真心感谢您们!

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] 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/.

[3] 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/.

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

[5] 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/.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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/.

[13] 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/.

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

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

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

[17] 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/.

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

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

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

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

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

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

[24] 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.

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

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

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

[28] 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/.

[29] 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.

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

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

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

[33] 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.

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

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

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

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

简历与科研成果

基本信息

汪浩港，男，汉族，1996 年 12 月出生，江苏省扬州人。

教育背景

2014 年 9 月 — 2018 年 6 月 中国矿业大学计算机科学与技术系　　本科

攻读硕士学位期间完成的学术成果

1. 软件著作权：人机物融合资源管理云平台（登记号：2020SR1657985），2020 年 11 月 26 日

攻读硕士学位期间参与的科研课题

1. 国家重点研发项目：软件定义的人机物融合云计算支撑技术与平台（2018YFB004805），2018 年 5 月-2021 年 4 月

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	MG1833067	所在院系	计算机科学与技术系	学位年度	2018
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	15605213809		作者 Email	whg19961229@gmail.com	
第一导师姓名	曹春 教授		导师电话	18951679203	

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）。