

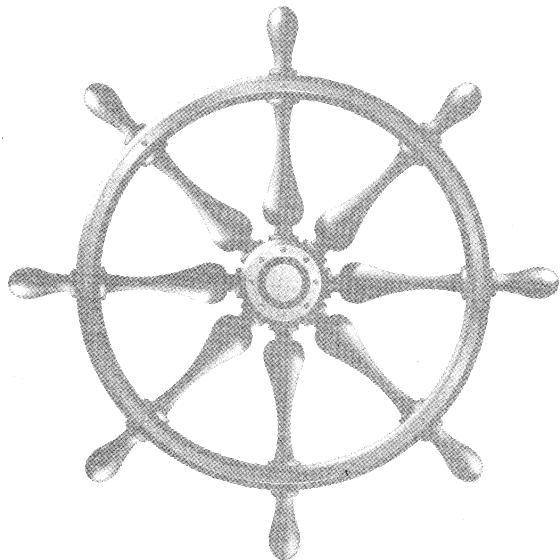
Kubernetes

权威指南

第2版

龚正 吴治辉 王伟
崔秀龙 闫健勇 / 等编著

从Docker到Kubernetes
实践全接触



电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Kubernetes 是由谷歌开源的 Docker 容器集群管理系统，为容器化的应用提供了资源调度、部署运行、服务发现、扩容及缩容等一整套功能。本书从一个开发者的角度去理解、分析和解决问题，囊括了 Kubernetes 入门、核心原理、实践指南、开发指导、高级案例、运维指南及源码分析等方面的内容，图文并茂、内容丰富、由浅入深、讲解全面；并围绕着生产环境中可能出现的问题，给出了大量的典型案例，比如安全问题、网络方案的选择、高可用性方案及 Trouble Shooting 技巧等，有很强的可借鉴性。

无论是对于软件工程师、测试工程师、运维工程师、软件架构师、技术经理，还是对于资深 IT 人士来说，本书都极具参考价值。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Kubernetes 权威指南：从 Docker 到 Kubernetes 实践全接触 / 龚正等编著. —2 版. —北京：电子工业出版社，2016.10

ISBN 978-7-121-29941-4

I. ①K… II. ①龚… III. ①Linux 操作系统—程序设计—指南 IV. ①TP316.85-62

中国版本图书馆 CIP 数据核字（2016）第 225554 号

策划编辑：张国霞

责任编辑：徐津平

印 刷：北京天宇星印刷厂

装 订：北京天宇星印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：32.5 字数：730 千字

版 次：2016 年 10 月第 1 版

印 次：2016 年 10 月第 1 次印刷

印 数：4000 册 定价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，
联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

第1版推荐序

经过作者们多年的实践经验积累及近一年的精心准备，本书终于与我们大家见面了。我有幸作为首批读者，提前见证和学习了在云时代引领业界技术方向的 Kubernetes 和 Docker 的最新动态。

从内容上讲，本书从一个开发者的角度去理解、分析和解决问题：从基础入门到架构原理，从运行机制到开发源码，再从系统运维到应用实践，讲解全面。本书图文并茂，内容丰富，由浅入深，对基本原理阐述清晰，对程序源码分析透彻，对实践经验体会深刻。

我认为本书值得推荐的原因有以下几点。

首先，作者的所有观点和经验，均是在多年建设、维护大型应用系统的过程中积累形成的。例如，读者通过学习书中的 Kubernetes 运维指南和高级应用实践案例章节的内容，不仅可以直接提高开发技能，还可以解决在实践过程中经常遇到的各种关键问题。书中的这些内容具有很高的借鉴和推广意义。

其次，通过大量的实例操作和详尽的源码解析，本书可以帮助读者进一步深刻理解 Kubernetes 的各种概念。例如书中“Java 访问 Kubernetes API”的几种方法，读者参照其中的案例，只要稍做修改，再结合实际的应用需求，就可以用于正在开发的项目中，达到事半功倍的效果，有利于有一定 Java 基础的专业人士快速学习 Kubernetes 的各种细节和实践操作。

再次，为了让初学者快速入门，本书配备了即时在线交流工具和专业后台技术支持团队。如果你在开发和应用的过程中遇到各类相关问题，均可直接联系该团队的开发支持专家。

最后，我们可以看到，容器化技术已经成为计算模型演化的一个开端，Kubernetes 作为谷歌开源的 Docker 容器集群管理技术，在这场新的技术革命中扮演着重要的角色。Kubernetes 正在被众多知名企业所采用，例如 RedHat、VMware、CoreOS 及腾讯等，因此，Kubernetes 站在了容器新技术变革的浪潮之巅，将具有不可预估的发展前景和商业价值。

如果你是初级程序员，那么你有必要好好学习本书；如果你正在 IT 领域进行高级进阶修炼，那你也有必要阅读本书。无论是架构师、开发者、运维人员，还是对容器技术比较好奇的读者，本书都是一本不可多得的带你从入门向高级进阶的精品书，值得大家选择！

初瑞

中国移动业务支撑中心高级经理

自序

我不知道你是如何获得这本书的，可能是在百度头条、网络广告、朋友圈中听说本书后购买的，也可能是某一天逛书店时，这本书恰好神奇地翻落书架，出现在你面前，让你想起一千多年前那个意外得到《太公兵法》的传奇少年，你觉得这是冥冥之中上天的恩赐，于是果断带走。不管怎样，我相信多年以后，这本书仍然值得你回忆。

Kubernetes 这个名字起源于古希腊，是舵手的意思，所以它的 Logo 既像一张渔网，又像一个罗盘。谷歌采用这个名字的一层深意就是：既然 Docker 把自己定位为驮着集装箱在大海上自在遨游的鲸鱼，那么谷歌就要以 Kubernetes 掌舵大航海时代的话语权，“捕获”和“指引”这条鲸鱼按照“主人”设定的路线巡游，确保谷歌倾力打造的新一代容器世界的宏伟蓝图顺利实现。

虽然 Kubernetes 自诞生至今才 1 年多，其第一个正式版本 Kubernetes 1.0 于 2015 年 7 月才发布，完全是个新生事物，但其影响力巨大，已经吸引了包括 IBM、惠普、微软、红帽、Intel、VMware、CoreOS、Docker、Mesosphere、Mirantis 等在内的众多业界巨头纷纷加入。红帽这个软件虚拟化领域的领导者之一，在容器技术方面已经完全“跟从”谷歌了，不仅把自家的第三代 OpenShift 产品的架构底层换成了 Docker+Kubernetes，还直接在其新一代容器操作系统 Atomic 内原生集成了 Kubernetes。

Kubernetes 是第一个将“一切以服务（Service）为中心，一切围绕服务运转”作为指导思想的创新型产品，它的功能和架构设计自始至终都遵循了这一指导思想，构建在 Kubernetes 上的系统不仅可以独立运行在物理机、虚拟机集群或者企业私有云上，也可以被托管在公有云中。Kubernetes 方案的另一个亮点是自动化，在 Kubernetes 的解决方案中，一个服务可以自我扩展、自我诊断，并且容易升级，在收到服务扩容的请求后，Kubernetes 会触发调度流程，最终在选定的目标节点上启动相应数量的服务实例副本，这些副本在启动成功后会自动加入负载均衡器中并生效，整个过程无须额外的人工操作。另外，Kubernetes 会定时巡查每个服务的所有实例的可用性，确保服务实例的数量始终保持为预期的数量，当它发现某个实例不可用时，会自动

重启该实例或者在其他节点重新调度、运行一个新实例，这样，一个复杂的过程无须人工干预即可全部自动化完成。试想一下，如果一个包括几十个节点且运行着几万个容器的复杂系统，其负载均衡、故障检测和故障修复等都需要人工介入进行处理，那将是多么难以想象。

通常我们会把 Kubernetes 看作 Docker 的上层架构，就好像 Java 与 J2EE 的关系一样：J2EE 是以 Java 为基础的企业级软件架构，而 Kubernetes 则以 Docker 为基础打造了一个云计算时代的全新分布式系统架构。但 Kubernetes 与 Docker 之间还存在着更为复杂的关系，从表面上看，似乎 Kubernetes 离不开 Docker，但实际上在 Kubernetes 的架构里，Docker 只是其目前支持的两种底层容器技术之一，另一个容器技术则是 Rocket，后者来源于 CoreOS 这个 Docker 昔日的“恋人”所推出的竞争产品。

Kubernetes 同时支持这两种互相竞争的容器技术，这是有深刻的历史原因的。快速发展的 Docker 打败了谷歌曾经名噪一时的开源容器技术 lmctfy，并迅速风靡世界。但是，作为一个已经对全球 IT 公司产生重要影响的技术，Docker 背后的容器标准的制定注定不可能被任何一个公司私有控制，于是就有了后来引发危机的 CoreOS 与 Docker 分手事件，其导火索是 CoreOS 撤开了 Docker，推出了与 Docker 相对抗的开源容器项目——Rocket，并动员一些知名 IT 公司成立委员会来试图主导容器技术的标准化，该分手事件愈演愈烈，最终导致 CoreOS “傍上”谷歌一起宣布“叛逃”Docker 阵营，共同发起了基于 CoreOS+Rocket+Kubernetes 的新项目 Tectonic。这让当时的 Docker 阵营和 Docker 粉丝们无比担心 Docker 的命运，不管最终鹿死谁手，容器技术分裂态势的加剧对所有牵涉其中的人来说都没有好处，于是 Linux 基金会出面调和矛盾，双方都退让一步，最终的结果是 Linux 基金会于 2015 年 6 月宣布成立开放容器技术项目（Open Container Project），谷歌、CoreOS 及 Docker 都加入了 OCP 项目。但通过查看 OCP 项目的成员名单，你会发现 Docker 在这个名单中只能算一个小角色了。OCP 的成立最终结束了这场让无数人揪心的“战争”，Docker 公司被迫放弃了自己的独家控制权。作为回报，Docker 的容器格式被 OCP 采纳为新标准的基础，并且由 Docker 负责起草 OCP 草案规范的初稿文档，当然这个“标准起草者”的角色也不是那么容易担当的，Docker 要提交自己的容器执行引擎的源码作为 OCP 项目的启动资源。

事到如今，我们再来看看当初 CoreOS 与谷歌的叛逃事件，从表面上看，谷歌貌似是被诱拐“出柜”的，但局里人都明白，谷歌才是这一系列事件背后的主谋，其不仅为当年失败的 lmctfy 报了一箭之仇，还重新掌控了容器技术的未来。容器标准之战大捷之后，谷歌进一步扩大了联盟并提高了自身影响力。2015 年 7 月，谷歌正式宣布加入 OpenStack 阵营，其目标是确保 Linux 容器及关联的容器管理技术 Kubernetes 能够被 OpenStack 生态圈所容纳，并且成为 OpenStack 平台上与 KVM 虚机一样的一等公民。谷歌加入 OpenStack 意味着对数据中心控制平面的争夺已经结束，以容器为代表的应用形态与以虚拟化为代表的系统形态将会完美融合于 OpenStack 之上，并与软件定义网络和软件定义存储一起统治下一代数据中心。

谷歌凭借着几十年大规模容器使用的丰富经验，步步为营，先是祭出 Kubernetes 这个神器，然后又掌控了容器技术的制定标准，最后又入驻 OpenStack 阵营全力将 Kubernetes 扶上位，谷歌这个 IT 界的领导者和创新者再次王者归来。我们都明白，在 IT 世界里只有那些被大公司掌控和推广的，同时被业界众多巨头都认可和支持的新技术才能生存和壮大下去。Kubernetes 就是当今 IT 界里符合要求且为数不多的热门技术之一，它的影响力可能长达十年，所以，我们每个 IT 人都有理由重视这门新技术。

谁能比别人领先一步掌握新技术，谁就在竞争中赢得了先机。惠普中国电信解决方案领域的资深专家团一起分工协作，并行研究，废寝忘食地合力撰写，在短短的 5 个月内完成了这部厚达 500 多页的 Kubernetes 权威指南。经过一年的高速发展，Kubernetes 先后发布了 1.1、1.2 和 1.3 版本，每个版本都带来了大量的新特性，能够处理的应用场景也越来越丰富。本书遵循从入门到精通的学习路线，全书共分为六大章节，涵盖了入门、实践指南、架构原理、开发指南、高级案例、运维指南和源码分析等内容，内容详实、图文并茂，几乎囊括了 Kubernetes 1.3 版本的方方面面，无论是对于软件工程师、测试工程师、运维工程师、软件架构师、技术经理，还是对于资深 IT 人士来说，本书都极具参考价值。

吴治辉

惠普公司系统架构师

目 录

第 1 章 Kubernetes 入门

1

1.1 Kubernetes 是什么	1
1.2 为什么要用 Kubernetes	4
1.3 从一个简单的例子开始	5
1.3.1 环境准备	6
1.3.2 启动 MySQL 服务	7
1.3.3 启动 Tomcat 应用	9
1.3.4 通过浏览器访问网页	11
1.4 Kubernetes 基本概念和术语	12
1.4.1 Master	12
1.4.2 Node	13
1.4.3 Pod	15
1.4.4 Label (标签)	19
1.4.5 Replication Controller (RC)	22
1.4.6 Deployment	25
1.4.7 Horizontal Pod Autoscaler (HPA)	27
1.4.8 Service (服务)	29
1.4.9 Volume (存储卷)	35

1.4.10 Persistent Volume	39
1.4.11 Namespace（命名空间）	40
1.4.12 Annotation（注解）	42
1.4.13 小结	42

第 2 章 Kubernetes 实践指南

43

2.1 Kubernetes 安装与配置	43
2.1.1 安装 Kubernetes	43
2.1.2 配置和启动 Kubernetes 服务	45
2.1.3 Kubernetes 集群的安全设置	51
2.1.4 Kubernetes 的版本升级	57
2.1.5 内网中的 Kubernetes 相关配置	57
2.1.6 Kubernetes 核心服务配置详解	58
2.1.7 Kubernetes 集群网络配置方案	72
2.2 kubectl 命令行工具用法详解	80
2.2.1 kubectl 用法概述	80
2.2.2 kubectl 子命令详解	82
2.2.3 kubectl 参数列表	84
2.2.4 kubectl 输出格式	84
2.2.5 kubectl 操作示例	86
2.3 Guestbook 示例：Hello World	87
2.3.1 创建 redis-master RC 和 Service	89
2.3.2 创建 redis-slave RC 和 Service	91
2.3.3 创建 frontend RC 和 Service	93
2.3.4 通过浏览器访问 frontend 页面	96
2.4 深入掌握 Pod	97
2.4.1 Pod 定义详解	97

2.4.2 Pod 的基本用法	102
2.4.3 静态 Pod	107
2.4.4 Pod 容器共享 Volume	108
2.4.5 Pod 的配置管理	110
2.4.6 Pod 生命周期和重启策略	123
2.4.7 Pod 健康检查	124
2.4.8 玩转 Pod 调度	126
2.4.9 Pod 的扩容和缩容	135
2.4.10 Pod 的滚动升级	139
2.5 深入掌握 Service	143
2.5.1 Service 定义详解	143
2.5.2 Service 基本用法	145
2.5.3 集群外部访问 Pod 或 Service	150
2.5.4 DNS 服务搭建指南	153
2.5.5 Ingress: HTTP 7 层路由机制	161

第 3 章 Kubernetes 核心原理

165

3.1 Kubernetes API Server 原理分析	165
3.1.1 Kubernetes API Server 概述	165
3.1.2 独特的 Kubernetes Proxy API 接口	168
3.1.3 集群功能模块之间的通信	169
3.2 Controller Manager 原理分析	170
3.2.1 Replication Controller	171
3.2.2 Node Controller	173
3.2.3 ResourceQuota Controller	174
3.2.4 Namespace Controller	176
3.2.5 Service Controller 与 Endpoint Controller	176

3.3 Scheduler 原理分析	177
3.4 kubelet 运行机制分析	181
3.4.1 节点管理	181
3.4.2 Pod 管理	182
3.4.3 容器健康检查	183
3.4.4 cAdvisor 资源监控	184
3.5 kube-proxy 运行机制分析	186
3.6 深入分析集群安全机制	190
3.6.1 API Server 认证	190
3.6.2 API Server 授权	192
3.6.3 Admission Control 准入控制	194
3.6.4 Service Account	195
3.6.5 Secret 秘密凭据	200
3.7 网络原理	203
3.7.1 Kubernetes 网络模型	203
3.7.2 Docker 的网络基础	205
3.7.3 Docker 的网络实现	217
3.7.4 Kubernetes 的网络实现	225
3.7.5 开源的网络组件	229
3.7.6 网络实战	234

第 4 章 Kubernetes 开发指南

247

4.1 REST 简述	247
4.2 Kubernetes API 详解	249
4.2.1 Kubernetes API 概述	249
4.2.2 API 版本	254
4.2.3 API 详细说明	254

4.2.4 API 响应说明	256
4.3 使用 Java 程序访问 Kubernetes API	258
4.3.1 Jersey	258
4.3.2 Fabric8	270
4.3.3 使用说明	271

第 5 章 Kubernetes 运维指南

292

5.1 Kubernetes 集群管理指南	292
5.1.1 Node 的管理	292
5.1.2 更新资源对象的 Label	294
5.1.3 Namespace: 集群环境共享与隔离	295
5.1.4 Kubernetes 资源管理	299
5.1.5 Kubernetes 集群高可用部署方案	333
5.1.6 Kubernetes 集群监控	343
5.1.7 kubelet 的垃圾回收 (GC) 机制	361
5.2 Kubernetes 高级案例	362
5.2.1 ElasticSearch 日志搜集查询和展现案例	362
5.2.2 Cassandra 集群部署案例	371
5.3 Trouble Shooting 指导	376
5.3.1 查看系统 Event 事件	377
5.3.2 查看容器日志	379
5.3.3 查看 Kubernetes 服务日志	379
5.3.4 常见问题	381
5.3.5 寻求帮助	384
5.4 Kubernetes v1.3 开发中的新功能	385
5.4.1 Pet Set (有状态的容器)	385
5.4.2 Init Container (初始化容器)	388
5.4.3 Cluster Federation (集群联邦)	391

第 6 章 Kubernetes 源码导读

396

6.1	Kubernetes 源码结构和编译步骤	396
6.2	kube-apiserver 进程源码分析	400
6.2.1	进程启动过程	400
6.2.2	关键代码分析	402
6.2.3	设计总结	417
6.3	kube-controller-manager 进程源码分析	420
6.3.1	进程启动过程	420
6.3.2	关键代码分析	423
6.3.3	设计总结	431
6.4	kube-scheduler 进程源码分析	433
6.4.1	进程启动过程	434
6.4.2	关键代码分析	438
6.4.3	设计总结	445
6.5	kubelet 进程源码分析	447
6.5.1	进程启动过程	447
6.5.2	关键代码分析	452
6.5.3	设计总结	475
6.6	kube-proxy 进程源码分析	476
6.6.1	进程启动过程	476
6.6.2	关键代码分析	478
6.6.3	设计总结	493
6.7	kubectl 进程源码分析	494
6.7.1	kubectl create 命令	495
6.7.2	rolling-update 命令	499
	后记	505

第 1 章

Kubernetes 入门

1.1 Kubernetes 是什么

Kubernetes 是什么？

首先，它是一个全新的基于容器技术的分布式架构领先方案。这个方案虽然还很新，但它是谷歌十几年以来大规模应用容器技术的经验积累和升华的一个重要成果。确切地说，Kubernetes 是谷歌严格保密十几年的秘密武器——Borg 的一个开源版本。Borg 是谷歌的一个久负盛名的内部使用的大规模集群管理系统，它基于容器技术，目的是实现资源管理的自动化，以及跨多个数据中心的资源利用率的最大化。十几年来，谷歌一直通过 Borg 系统管理着数量庞大的应用程序集群。由于谷歌员工都签署了保密协议，即便离职也不能泄露 Borg 的内部设计，所以外界一直无法了解关于它的更多信息。直到 2015 年 4 月，传闻许久的 Borg 论文伴随 Kubernetes 的高调宣传被谷歌首次公开，大家才得以了解它的更多内幕。正是由于站在 Borg 这个前辈的肩膀上，吸取了 Borg 过去十年间的经验与教训，所以 Kubernetes 一经开源就一鸣惊人，并迅速称霸了容器技术领域。

其次，如果我们的系统设计遵循了 Kubernetes 的设计思想，那么传统系统架构中那些和业务没有多大关系的底层代码或功能模块，都可以立刻从我们的视线中消失，我们不必再费心于负载均衡器的选型和部署实施问题，不必再考虑引入或自己开发一个复杂的服务治理框架，不必再头疼于服务监控和故障处理模块的开发。总之，使用 Kubernetes 提供的解决方案，我们不仅节省了不少于 30% 的开发成本，同时可以将精力更加集中于业务本身，而且由于 Kubernetes 提供了强大的自动化机制，所以系统后期的运维难度和运维成本大幅度降低。

然后，Kubernetes 是一个开放的开发平台。与 J2EE 不同，它不局限于任何一种语言，没有

限定任何编程接口，所以不论是用 Java、Go、C++还是用 Python 编写的服务，都可以毫无困难地映射为 Kubernetes 的 Service，并通过标准的 TCP 通信协议进行交互。此外，由于 Kubernetes 平台对现有的编程语言、编程框架、中间件没有任何侵入性，因此现有的系统也很容易改造升级并迁移到 Kubernetes 平台上。

最后，Kubernetes 是一个完备的分布式系统支撑平台。Kubernetes 具有完备的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建智能负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制，以及多粒度的资源配额管理能力。同时，Kubernetes 提供了完善的管理工具，这些工具涵盖了包括开发、部署测试、运维监控在内的各个环节。因此，Kubernetes 是一个全新的基于容器技术的分布式架构解决方案，并且是一个一站式的完备的分布式系统开发和支撑平台。

在正式开始本章的 Hello World 之旅之前，我们首先要学习 Kubernetes 的一些基本知识，这样我们才能理解 Kubernetes 提供的解决方案。

在 Kubernetes 中，Service（服务）是分布式集群架构的核心，一个 Service 对象拥有如下关键特征。

- ◎ 拥有一个唯一指定的名字（比如 mysql-server）。
- ◎ 拥有一个虚拟 IP（Cluster IP、Service IP 或 VIP）和端口号。
- ◎ 能够提供某种远程服务能力。
- ◎ 被映射到了提供这种服务能力的一组容器应用上。

Service 的服务进程目前都基于 Socket 通信方式对外提供服务，比如 Redis、Memcache、MySQL、Web Server，或者是实现了某个具体业务的一个特定的 TCP Server 进程。虽然一个 Service 通常由多个相关的服务进程来提供服务，每个服务进程都有一个独立的 Endpoint (IP+Port) 访问点，但 Kubernetes 能够让我们通过 Service（虚拟 Cluster IP +Service Port）连接到指定的 Service 上。有了 Kubernetes 内建的透明负载均衡和故障恢复机制，不管后端有多少服务进程，也不管某个服务进程是否会由于发生故障而重新部署到其他机器，都不会影响到我们对服务的正常调用。更重要的是这个 Service 本身一旦创建就不再变化，这意味着，在 Kubernetes 集群中，我们再也不用为了服务的 IP 地址变来变去的问题而头疼了。

容器提供了强大的隔离功能，所以有必要把为 Service 提供服务的这组进程放入容器中进行隔离。为此，Kubernetes 设计了 Pod 对象，将每个服务进程包装到相应的 Pod 中，使其成为 Pod 中运行的一个容器（Container）。为了建立 Service 和 Pod 间的关联关系，Kubernetes 首先给每个 Pod 贴上一个标签（Label），给运行 MySQL 的 Pod 贴上 name=mysql 标签，给运行 PHP 的

Pod 贴上 name=php 标签，然后给相应的 Service 定义标签选择器（Label Selector），比如 MySQL Service 的标签选择器的选择条件为 name=mysql，意为该 Service 要作用于所有包含 name=mysql Label 的 Pod 上。这样一来，就巧妙地解决了 Service 与 Pod 的关联问题。

说到 Pod，我们这里先简单介绍其概念。首先，Pod 运行在一个我们称之为节点（Node）的环境中，这个节点既可以是物理机，也可以是私有云或者公有云中的一个虚拟机，通常在一个节点上运行几百个 Pod；其次，每个 Pod 里运行着一个特殊的被称之为 Pause 的容器，其他容器则为业务容器，这些业务容器共享 Pause 容器的网络栈和 Volume 挂载卷，因此它们之间的通信和数据交换更为高效，在设计时我们可以充分利用这一特性将一组密切相关的服务进程放入同一个 Pod 中；最后，需要注意的是，并不是每个 Pod 和它里面运行的容器都能“映射”到一个 Service 上，只有那些提供服务（无论是对内还是对外）的一组 Pod 才会被“映射”成一个服务。

在集群管理方面，Kubernetes 将集群中的机器划分为一个 Master 节点和一群工作节点（Node）。其中，在 Master 节点上运行着集群管理相关的一组进程 kube-apiserver、kube-controller-manager 和 kube-scheduler，这些进程实现了整个集群的资源管理、Pod 调度、弹性伸缩、安全控制、系统监控和纠错等管理功能，并且都是全自动完成的。Node 作为集群中的工作节点，运行真正的应用程序，在 Node 上 Kubernetes 管理的最小运行单元是 Pod。Node 上运行着 Kubernetes 的 kubelet、kube-proxy 服务进程，这些服务进程负责 Pod 的创建、启动、监控、重启、销毁，以及实现软件模式的负载均衡器。

最后，我们再来看看传统的 IT 系统中服务扩容和服务升级这两个难题，以及 Kubernetes 所提供的全新解决思路。服务的扩容涉及资源分配（选择哪个节点进行扩容）、实例部署和启动等环节，在一个复杂的业务系统中，这两个问题基本上靠人工一步步操作才得以完成，费时费力又难以保证实施质量。

在 Kubernetes 集群中，你只需为需要扩容的 Service 关联的 Pod 创建一个 Replication Controller（简称 RC），则该 Service 的扩容以至于后来的 Service 升级等头疼问题都迎刃而解。在一个 RC 定义文件中包括以下 3 个关键信息。

- ◎ 目标 Pod 的定义。
- ◎ 目标 Pod 需要运行的副本数量（Replicas）。
- ◎ 要监控的目标 Pod 的标签（Label）。

在创建好 RC（系统将自动创建好 Pod）后，Kubernetes 会通过 RC 中定义的 Label 筛选出对应的 Pod 实例并实时监控其状态和数量，如果实例数量少于定义的副本数量（Replicas），则会根据 RC 中定义的 Pod 模板来创建一个新的 Pod，然后将此 Pod 调度到合适的 Node 上启动运行，直到 Pod 实例的数量达到预定目标。这个过程完全是自动化的，无须人工干预。有了 RC，

服务的扩容就变成了一个纯粹的简单数字游戏了，只要修改 RC 中的副本数量即可。后续的 Service 升级也将通过修改 RC 来自动完成。

以将在第 2 章介绍的 PHP+Redis 留言板应用为例，只要为 PHP 留言板程序（frontend）创建一个有 3 个副本的 RC+Service，为 Redis 读写分离集群创建两个 RC：写节点（redis-master）创建一个单副本的 RC+Service，读节点（redis-slaver）创建一个有两个副本的 RC+Service，就可以分分钟完成整个集群的搭建过程了，是不是很简单？

1.2 为什么要用 Kubernetes

使用 Kubernetes 的理由很多，最根本的一个理由就是：IT 从来都是一个由新技术驱动的行业。

Docker 这个新兴的容器化技术当前已经被很多公司所采用，其从单机走向集群已成为必然，而云计算的蓬勃发展正在加速这一进程。Kubernetes 作为当前唯一被业界广泛认可和看好的 Docker 分布式系统解决方案，可以预见，在未来几年内，会有大量的新系统选择它，不管这些系统是运行在企业本地服务器上还是被托管到公有云上。

使用了 Kubernetes 又会收获哪些好处呢？

首先，最直接的感受就是我们可以“轻装上阵”地开发复杂系统了。以前动不动就需要十几个人而且团队里需要不少技术达人一起分工协作才能设计实现和运维的分布式系统，在采用 Kubernetes 解决方案之后，只需一个精悍的小团队就能轻松应对。在这个团队里，一名架构师专注于系统中“服务组件”的提炼，几名开发工程师专注于业务代码的开发，一名系统兼运维工程师负责 Kubernetes 的部署和运维，从此再也不用“996”了，这并不是因为我们少做了什么，而是因为 Kubernetes 已经帮我们做了很多。

其次，使用 Kubernetes 就是在全面拥抱微服务架构。微服务架构的核心是将一个巨大的单体应用分解为很多小的互相连接的微服务，一个微服务背后可能有多个实例副本在支撑，副本的数量可能会随着系统的负荷变化而进行调整，内嵌的负载均衡器在这里发挥了重要作用。微服务架构使得每个服务都可以由专门的开发团队来开发，开发者可以自由选择开发技术，这对于大规模团队来说很有价值，另外每个微服务独立开发、升级、扩展，因此系统具备很高的稳定性和快速迭代进化能力。谷歌、亚马逊、eBay、NetFlix 等众多大型互联网公司都采用了微服务架构，此次谷歌更是将微服务架构的基础设施直接打包到 Kubernetes 解决方案中，让我们有机会直接应用微服务架构解决复杂业务系统的架构问题。

然后，我们的系统可以随时随地整体“搬迁”到公有云上。Kubernetes 最初的目标就是运

行在谷歌自家的公有云 GCE 中，未来会支持更多的公有云及基于 OpenStack 的私有云。同时，在 Kubernetes 的架构方案中，底层网络的细节完全被屏蔽，基于服务的 Cluster IP 甚至都无须我们改变运行期的配置文件，就能将系统从物理机环境中无缝迁移到公有云中，或者在服务高峰期将部分服务对应的 Pod 副本放入公有云中以提升系统的吞吐量，不仅节省了公司的硬件投入，还大大改善了客户体验。我们所熟知的铁道部的 12306 购票系统，在春节高峰期就租用了阿里云进行分流。

最后，Kubernetes 系统架构具备了超强的横向扩容能力。对于互联网公司来说，用户规模就等价于资产，谁拥有更多的用户，谁就能在竞争中胜出，因此超强的横向扩容能力是互联网业务系统的关键指标之一。不用修改代码，一个 Kubernetes 集群即可从只包含几个 Node 的小集群平滑扩展到拥有上百个 Node 的大规模集群，我们利用 Kubernetes 提供的工具，甚至可以在线完成集群扩容。只要我们的微服务设计得好，结合硬件或者公有云资源的线性增加，系统就能够承受大量用户并发访问所带来的巨大压力。

1.3 从一个简单的例子开始

考虑到本书第 1 版中的 PHP+Redis 留言板的 Hello World 例子对于绝大多数刚接触 Kubernetes 的人来说比较复杂，难以顺利上手和实践，所以我们在此将这个例子替换成一个简单得多的 Java Web 应用，可以让新手快速上手和实践。

此 Java Web 应用的结构比较简单，是一个运行在 Tomcat 里的 Web App，如图 1.1 所示，JSP 页面通过 JDBC 直接访问 MySQL 数据库并展示数据。为了演示和简化的目的，只要程序正确连接到了数据库上，它就会自动完成对应的 Table 的创建与初始化数据的准备工作。所以，当我们通过浏览器访问此应用的时候，就会显示一个表格的页面，数据则来自数据库。

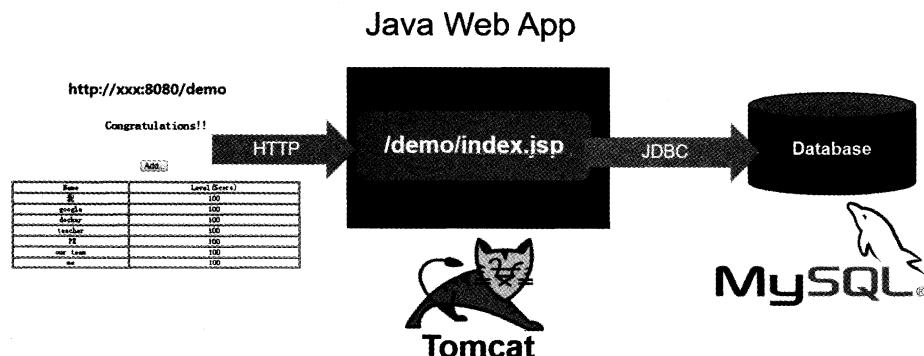


图 1.1 Java Web 应用的架构组成

此应用需要启动两个容器：Web App 容器和 MySQL 容器，并且 Web App 容器需要访问 MySQL 容器。在 Docker 时代，假设我们在一个宿主机上启动了这两个容器，则我们需要把 MySQL 容器的 IP 地址通过环境变量的方式注入 Web App 容器里；同时，需要将 Web App 容器的 8080 端口映射到宿主机的 8080 端口，以便能在外部访问。在本章的这个例子里，我们看看在 Kubernetes 时代是如何完成这个目标的。

1.3.1 环境准备

首先，我们开始准备 Kubernetes 的安装和相关镜像下载，本书建议采用 VirtualBox 或者 VMware Workstation 在本机虚拟一个 64 位的 CentOS 7 虚拟机作为学习环境，虚拟机采用 NAT 的网络模式以便能够连接外网，然后按照以下步骤快速安装 Kubernetes。

(1) 关闭 CentOS 自带的防火墙服务：

```
# systemctl disable firewalld  
# systemctl stop firewalld
```

(2) 安装 etcd 和 Kubernetes 软件（会自动安装 Docker 软件）：

```
# yum install -y etcd kubernetes
```

(3) 安装好软件后，修改两个配置文件（其他配置文件使用系统默认的配置参数即可）。

◎ Docker 配置文件为/etc/sysconfig/docker，其中 OPTIONS 的内容设置为：

```
OPTIONS='--selinux-enabled=false --insecure-registry gcr.io'
```

◎ Kubernetes apiserver 配置文件为/etc/kubernetes/apiserver，把--admission_control 参数中的 ServiceAccount 删除。

(4) 按顺序启动所有的服务：

```
# systemctl start etcd  
# systemctl start docker  
# systemctl start kube-apiserver  
# systemctl start kube-controller-manager  
# systemctl start kube-scheduler  
# systemctl start kubelet  
# systemctl start kube-proxy
```

至此，一个单机版的 Kubernetes 集群环境就安装启动完成了。

接下来，我们可以在这个单机版的 Kubernetes 集群中上手练习了。

注：本书示例中的 Docker 镜像下载地址为 <https://hub.docker.com/u/kubeguide/>。

1.3.2 启动 MySQL 服务

首先为 MySQL 服务创建一个 RC 定义文件：mysql-rc.yaml，下面给出了该文件的完整内容和解释，如图 1.2 所示。

```

apiVersion: v1
kind: ReplicationController   副本控制器 RC
metadata:
  name: mysql               RC 的名称，全局唯一
spec:
  replicas: 1                Pod 副本期待数量
  selector:
    app: mysql               符合目标的 Pod 拥有此标签
  template:
    metadata:
      labels:
        app: mysql             根据此模板创建 Pod 的副本（实例）
      spec:
        containers:
          - name: mysql         Pod 内容器的定义部分
            image: mysql         容器的名称
            ports:
              - containerPort: 3306 容器暴露的端口号
            env:
              - name: MYSQL_ROOT_PASSWORD
                value: "123456"     注入到容器内的环境变量

```

图 1.2 RC 的定义和解说图

yaml 定义文件中的 kind 属性，用来表明此资源对象的类型，比如这里的值为“ReplicationController”，表示这是一个 RC；spec 一节中是 RC 的相关属性定义，比如 spec.selector 是 RC 的 Pod 标签（Label）选择器，即监控和管理拥有这些标签的 Pod 实例，确保当前集群上始终有且仅有 replicas 个 Pod 实例在运行，这里我们设置 replicas=1 表示只能运行一个 MySQL Pod 实例。当集群中运行的 Pod 数量小于 replicas 时，RC 会根据 spec.template 一节中定义的 Pod 模板来生成一个新的 Pod 实例，spec.template.metadata.labels 指定了该 Pod 的标签，需要特别注意的是：这里的 labels 必须匹配之前的 spec.selector，否则此 RC 每次创建了一个无法匹配 Label 的 Pod，就会不停地尝试创建新的 Pod，最终陷入“只为他人做嫁衣”的悲惨世界中，永无翻身之日。

创建好 redis-master-controller.yaml 文件以后，为了将它发布到 Kubernetes 集群中，我们在 Master 节点执行命令：

```
# kubectl create -f mysql-rc.yaml
replicationcontroller "mysql" created
```

接下来，我们用 kubectl 命令查看刚刚创建的 RC：

```
# kubectl get rc
NAME      DESIRED   CURRENT   AGE
mysql     1          1         7m
```

查看 Pod 的创建情况时，可以运行下面的命令：

```
# kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
mysql-c95jc 1/1     Running   0          9m
```

我们看到一个名为 mysql-xxxxx 的 Pod 实例，这是 Kubernetes 根据 mysql 这个 RC 的定义自动创建的 Pod。由于 Pod 的调度和创建需要花费一定的时间，比如需要一定的时间来确定调度到哪个节点上，以及下载 Pod 里容器的镜像需要一段时间，所以一开始我们看到 Pod 的状态将显示为 Pending。当 Pod 成功创建完成以后，状态最终会被更新为 Running。

我们通过 docker ps 指令查看正在运行的容器，发现提供 MySQL 服务的 Pod 容器已经创建并正常运行了，此外，你会发现 MySQL Pod 对应的容器还多创建了一个来自谷歌的 pause 容器，这就是 Pod 的“根容器”，详见 1.4.3 节的说明。

```
# docker ps | grep mysql
72ca992535b4 mysql
"docker-entrypoint.sh" 12 minutes ago Up 12 minutes
k8s_mysql.86dc506e_mysql-c95jc_default_511d6705-5051-11e6-a9d8-000c29ed42c1_9f89d0b4
  76c1790aad27 gcr.io/google_containers/pause-amd64:3.0
"/pause" 12 minutes ago Up 12 minutes
k8s POD.16b20365_mysql-c95jc_default_511d6705-5051-11e6-a9d8-000c29ed42c1_28520aba
```

最后，我们创建一个与之关联的 Kubernetes Service——MySQL 的定义文件（文件名为 mysql-svc.yaml），完整的内容和解释如图 1.3 所示。

```
apiVersion: v1
kind: Service           表明是 Kubernetes Service
metadata:
  name: mysql          Service 的全局唯一名称
spec:
  ports:
    - port: 3306        Service 提供服务的端口号
  selector:
    app: mysql         Service 对应的 Pod 拥有这里定义的标签
```

图 1.3 Service 的定义和解说图

其中， metadata.name 是 Service 的服务名（ServiceName）； port 属性则定义了 Service 的虚端口； spec.selector 确定了哪些 Pod 副本（实例）对应到本服务。类似地，我们通过 kubectl create 命令创建 Service 对象。

运行 kubectl 命令，创建 service:

```
# kubectl create -f mysql-svc.yaml
service "mysql" created
```

运行 kubectl 命令，可以查看到刚刚创建的 service:

```
# kubectl get svc
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
mysql    169.169.253.143  <none>        3306/TCP    48s
```

注意到 MySQL 服务被分配了一个值为 169.169.253.143 的 Cluster IP 地址，这是一个虚地址，随后，Kubernetes 集群中其他新创建的 Pod 就可以通过 Service 的 Cluster IP+端口号 6379 来连接和访问它了。

在通常情况下，Cluster IP 是在 Service 创建后由 Kubernetes 系统自动分配的，其他 Pod 无法预先知道某个 Service 的 Cluster IP 地址，因此需要一个服务发现机制来找到这个服务。为此，最初的时候，Kubernetes 巧妙地使用了 Linux 环境变量（Environment Variable）来解决这个问题，后面会详细说明其机制。现在我们只需知道，根据 Service 的唯一名字，容器可以从环境变量中获取到 Service 对应的 Cluster IP 地址和端口，从而发起 TCP/IP 连接请求了。

1.3.3 启动 Tomcat 应用

上面我们定义和启动了 MySQL 服务，接下来我们采用同样的步骤，完成 Tomcat 应用的启动过程。首先，创建对应的 RC 文件 myweb-rc.yaml，内容如下：

```
kind: ReplicationController
metadata:
  name: myweb
spec:
  replicas: 5
  selector:
    app: myweb
  template:
    metadata:
      labels:
        app: myweb
    spec:
      containers:
        - name: myweb
          image: kubeguide/tomcat-app:v1
          ports:
            - containerPort: 8080
          env:
            - name: MYSQL_SERVICE_HOST
```

```
    value: 'mysql'  
- name: MYSQL_SERVICE_PORT  
  value: '3306'
```

注意到上面 RC 对应的 Tomcat 容器里引用了 `MYSQL_SERVICE_HOST=mysql` 这个环境变量，而“mysql”恰好是我们之前定义的 MySQL 服务的服务名，运行下面的命令，完成 RC 的创建和验证工作：

```
# kubectl create -f myweb-rc.yaml  
replicationcontroller "myweb" created
```

```
# kubectl get pods  
NAME          READY   STATUS    RESTARTS   AGE  
mysql-c95jc   1/1     Running   0          2h  
myweb-g9pmm   1/1     Running   0          3s
```

最后，创建对应的 Service。以下是完整的 yaml 定义文件（`myweb-svc.yaml`）：

```
apiVersion: v1  
kind: Service  
metadata:  
  name: myweb  
spec:  
  type: NodePort  
  ports:  
    - port: 8080  
      nodePort: 30001  
  selector:  
    app: myweb
```

注意 `type=NodePort` 和 `nodePort=30001` 的两个属性，表明此 Service 开启了 NodePort 方式的外网访问模式，在 Kubernetes 集群之外，比如在本机的浏览器里，可以通过 30001 这个端口访问 `myweb`（对应到 8080 的虚端口上）。

运行 `kubectl create` 命令进行创建：

```
# kubectl create -f myweb-svc.yaml  
You have exposed your service on an external port on all nodes in your  
cluster. If you want to expose this service to the external internet, you may  
need to set up firewall rules for the service port(s) (tcp:30001) to serve traffic.  
See http://releases.k8s.io/release-1.3/docs/user-guide/services-firewalls.md  
for more details.  
service "myweb" created
```

我们看到上面有提示信息，意思是需要把 30001 这个端口在防火墙上打开，以便外部的访问能穿过防火墙。

运行 `kubectl` 命令，查看创建的 Service：

```
# kubectl get services
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
mysql     169.169.253.143 <none>        3306/TCP      44m
myweb     169.169.149.215 <nodes>       8080/TCP      4m
kubernetes 169.169.0.1    <none>        443/TCP      16d
```

至此，我们的第1个Kubernetes例子搭建完成了，在下一节中我们验证结果。

1.3.4 通过浏览器访问网页

经过上面的几个步骤，我们终于成功实现了Kubernetes上第1个例子的部署搭建工作。现在一起见证成果吧，在你的笔记本上打开浏览器，输入`http://虚拟机IP:30001/demo/`。

比如虚机IP为192.168.18.131（可以通过`#ip a`命令进行查询），在浏览器里输入地址`http://192.168.18.131:30001/demo/`后，看到了如图1.4所示的网页界面，那么恭喜你，之前的努力没有白费，顺利闯关成功！

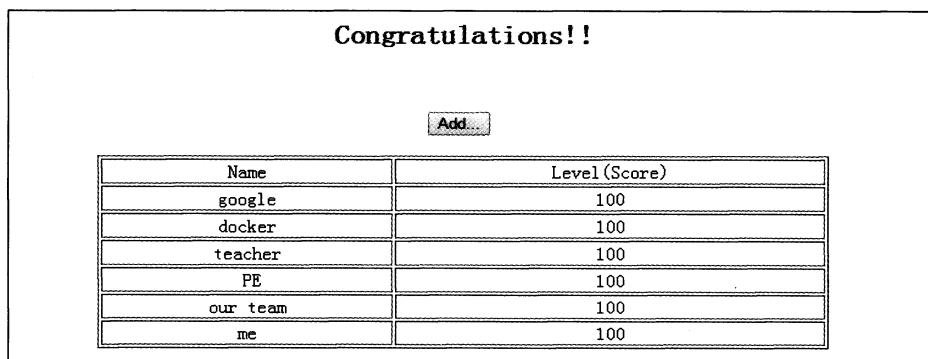


图1.4 通过浏览器访问Tomcat应用

如果看不到这个网页，那么可能有几个原因：比如防火墙的问题，无法访问30001端口，或者因为你是通过代理上网的，浏览器错把虚拟机的IP地址当成远程地址了。可以在虚拟机上直接运行`curl localhost:30001`来验证此端口是否能被访问，如果还是不能访问，那么这肯定不是机器的问题……

接下来可以尝试单击“Add...”按钮添加一条记录并提交，如图1.5所示，提交以后，数据就被写入MySQL数据库中了。

至此，我们终于完成了Kubernetes上的Tomcat例子，这个例子并不是很复杂。我们也看到，相对于传统的分布式应用的部署方式，在Kubernetes之上我们仅仅通过一些很容易理解的配置文件和相关的简单命令就完成了对整个集群的部署，这让我们惊诧于Kubernetes的创新和强大。

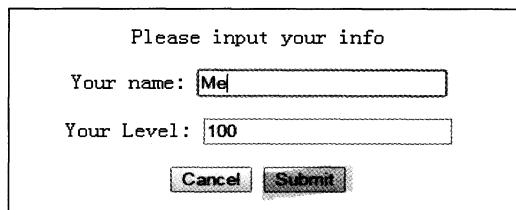


图 1.5 在留言板网页添加新的留言

下一节，我们将开始对 Kubernetes 中的基本概念和术语进行全面学习，在这之前，读者可以继续研究下这个例子里的一些拓展内容，如下所述。

- ◎ 研究 RC、Service 等文件的格式。
- ◎ 熟悉 kubectl 的子命令。
- ◎ 手工停止某个 Service 对应的容器进程，然后观察有什么现象发生。
- ◎ 修改 RC 文件，改变副本数量，重新发布，观察结果。

1.4 Kubernetes 基本概念和术语

Kubernetes 中的大部分概念如 Node、Pod、Replication Controller、Service 等都可以看作一种“资源对象”，几乎所有的资源对象都可以通过 Kubernetes 提供的 kubectl 工具（或者 API 编程调用）执行增、删、改、查等操作并将其保存在 etcd 中持久化存储。从这个角度来看，Kubernetes 其实是一个高度自动化的资源控制系统，它通过跟踪对比 etcd 库里保存的“资源期望状态”与当前环境中的“实际资源状态”的差异来实现自动控制和自动纠错的高级功能。

1.4.1 Master

Kubernetes 里的 Master 指的是集群控制节点，每个 Kubernetes 集群里需要有一个 Master 节点来负责整个集群的管理和控制，基本上 Kubernetes 所有的控制命令都是发给它，它来负责具体的执行过程，我们后面所有执行的命令基本都是在 Master 节点上运行的。Master 节点通常会占据一个独立的 X86 服务器（或者一个虚拟机），一个主要的原因是它太重要了，它是整个集群的“首脑”，如果它宕机或者不可用，那么我们所有的控制命令都将失效。

Master 节点上运行着以下一组关键进程。

- ◎ Kubernetes API Server (kube-apiserver)，提供了 HTTP Rest 接口的关键服务进程，是 Kubernetes 里所有资源的增、删、改、查等操作的唯一入口，也是集群控制的入口进程。

- ◎ Kubernetes Controller Manager (kube-controller-manager)，Kubernetes里所有资源对象的自动化控制中心，可以理解为资源对象的“大总管”。
- ◎ Kubernetes Scheduler (kube-scheduler)，负责资源调度（Pod 调度）的进程，相当于公交公司的“调度室”。

其实 Master 节点上往往还启动了一个 etcd Server 进程，因为 Kubernetes 里的所有资源对象的数据全部是保存在 etcd 中的。

1.4.2 Node

除了 Master，Kubernetes 集群中的其他机器被称为 Node 节点，在较早的版本中也被称为 Minion。与 Master 一样，Node 节点可以是一台物理主机，也可以是一台虚拟机。Node 节点才是 Kubernetes 集群中的工作负载节点，每个 Node 都会被 Master 分配一些工作负载（Docker 容器），当某个 Node 宕机时，其上的工作负载会被 Master 自动转移到其他节点上去。

每个 Node 节点上都运行着以下一组关键进程。

- ◎ kubelet：负责 Pod 对应的容器的创建、启停等任务，同时与 Master 节点密切协作，实现集群管理的基本功能。
- ◎ kube-proxy：实现 Kubernetes Service 的通信与负载均衡机制的重要组件。
- ◎ Docker Engine (docker)：Docker 引擎，负责本机的容器创建和管理工作。

Node 节点可以在运行期间动态增加到 Kubernetes 集群中，前提是这个节点上已经正确安装、配置和启动了上述关键进程，在默认情况下 kubelet 会向 Master 注册自己，这也是 Kubernetes 推荐的 Node 管理方式。一旦 Node 被纳入集群管理范围，kubelet 进程就会定时向 Master 节点汇报自身的情报，例如操作系统、Docker 版本、机器的 CPU 和内存情况，以及之前有哪些 Pod 在运行等，这样 Master 可以获知每个 Node 的资源使用情况，并实现高效均衡的资源调度策略。而某个 Node 超过指定时间不上报信息时，会被 Master 判定为“失联”，Node 的状态被标记为不可用（Not Ready），随后 Master 会触发“工作负载大转移”的自动流程。

我们可以执行下述命令查看集群中有多少个 Node：

```
# kubectl get nodes
NAME           STATUS    AGE
kubernetes-minion1   Ready    2d
```

然后，通过 kubectl describe node <node_name> 来查看某个 Node 的详细信息：

```
$ kubectl describe node kubernetes-minion1
```

```
Name:      k8s-node-1
```

```
Labels:    beta.kubernetes.io/arch=amd64
          beta.kubernetes.io/os=linux
          kubernetes.io/hostname=k8s-node-1
Taints:     <none>
CreationTimestamp:  Wed, 06 Jul 2016 11:46:41 +0800
Phase:
Conditions:
  Type      Status  LastHeartbeatTime           Value
  OutOfDisk False   Sat, 09 Jul 2016 08:17:39 +0800   ...
  MemoryPressure False  Sat, 09 Jul 2016 08:17:39 +0800   ...
  Ready      True    Sat, 09 Jul 2016 08:17:39 +0800   ...
Addresses:  192.168.18.131,192.168.18.131
Capacity:
  alpha.kubernetes.io/nvidia-gpu:  0
  cpu:                      4
  memory:                   1868692Ki
  pods:                     110
Allocatable:
  alpha.kubernetes.io/nvidia-gpu:  0
  cpu:                      4
  memory:                   1868692Ki
  pods:                     110
System Info:
  Machine ID:       6e4e2af2afeb42b9aac47d866aa56ca0
  System UUID:      564D63D3-9664-3393-A3DC-9CD424ED42C1
  Boot ID:          b0c34f9f-76ab-478e-9771-bd4fe6e98880
  Kernel Version:   3.10.0-327.22.2.el7.x86_64
  OS Image:         CentOS Linux 7 (Core)
  Operating System: linux
  Architecture:    amd64
  Container Runtime Version: docker://1.11.2
  Kubelet Version:  v1.3.0
  Kube-Proxy Version: v1.3.0
  ExternalID:       k8s-node-1
Non-terminated Pods: (1 in total)
  Namespace  Name            CPU Requests  CPU Limits  Memory Limits
  kube-system kube-dns-v11-wxdhf  310m (7%)  310m (7%)  170Mi (9%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted. More info:
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  310m (7%)    310m (7%)  170Mi (9%)    170Mi (9%)
No events.
```

上述命令展示了 Node 的如下关键信息。

- ◎ Node 基本信息：名称、标签、创建时间等。
- ◎ Node 当前的运行状态，Node 启动以后会做一系列的自检工作，比如磁盘是否满了，如果满了就标注 OutOfDisk=True，否则继续检查内存是否不足（MemoryPressure=True），最后一切正常，就切换为 Ready 状态（Ready=True），这种情况表示 Node 处于健康状态，可以在其上创建新的 Pod。
- ◎ Node 的主机地址与主机名。
- ◎ Node 上的资源总量：描述 Node 可用的系统资源，包括 CPU、内存数量、最大可调度 Pod 数量等，注意到目前 Kubernetes 已经实验性地支持 GPU 资源分配了（alpha.kubernetes.io/nvidia-gpu=0）。
- ◎ Node 可分配资源量：描述 Node 当前可用于分配的资源量。
- ◎ 主机系统信息：包括主机的唯一标识 UUID、Linux kernel 版本号、操作系统类型与版本、Kubernetes 版本号、kubelet 与 kube-proxy 的版本号等。
- ◎ 当前正在运行的 Pod 列表概要信息。
- ◎ 已分配的资源使用概要信息，例如资源申请的最低、最大允许使用量占系统总量的百分比。
- ◎ Node 相关的 Event 信息。

1.4.3 Pod

Pod 是 Kubernetes 的最重要也最基本的概念，如图 1.6 所示是 Pod 的组成示意图，我们看到每个 Pod 都有一个特殊的被称为“根容器”的 Pause 容器。Pause 容器对应的镜像属于 Kubernetes 平台的一部分，除了 Pause 容器，每个 Pod 还包含一个或多个紧密相关的用户业务容器。

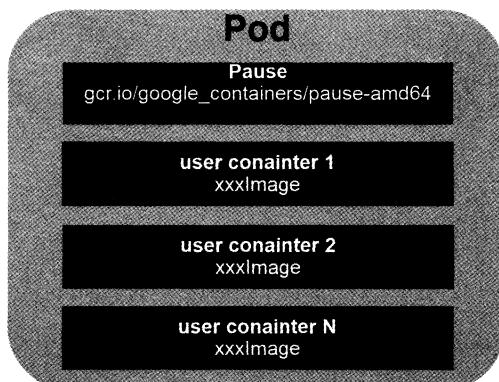


图 1.6 Pod 的组成与容器的关系

为什么 Kubernetes 会设计出一个全新的 Pod 的概念并且 Pod 有这样特殊的组成结构？

原因之一：在一组容器作为一个单元的情况下，我们难以对“整体”简单地进行判断及有效地进行行动。比如，一个容器死亡了，此时算是整体死亡么？是 N/M 的死亡率么？引入业务无关并且不易死亡的 Pause 容器作为 Pod 的根容器，以它的状态代表整个容器组的状态，就简单、巧妙地解决了这个难题。

原因之二：Pod 里的多个业务容器共享 Pause 容器的 IP，共享 Pause 容器挂接的 Volume，这样既简化了密切关联的业务容器之间的通信问题，也很好地解决了它们之间的文件共享问题。

Kubernetes 为每个 Pod 都分配了唯一的 IP 地址，称之为 Pod IP，一个 Pod 里的多个容器共享 Pod IP 地址。Kubernetes 要求底层网络支持集群内任意两个 Pod 之间的 TCP/IP 直接通信，这通常采用虚拟二层网络技术来实现，例如 Flannel、Openvswitch 等，因此我们需要牢记一点：在 Kubernetes 里，一个 Pod 里的容器与另外主机上的 Pod 容器能够直接通信。

Pod 其实有两种类型：普通的 Pod 及静态 Pod (static Pod)，后者比较特殊，它并不存放在 Kubernetes 的 etcd 存储里，而是存放在某个具体的 Node 上的一个具体文件中，并且只在此 Node 上启动运行。而普通的 Pod 一旦被创建，就会被放入到 etcd 中存储，随后会被 Kubernetes Master 调度到某个具体的 Node 上并进行绑定 (Binding)，随后该 Pod 被对应的 Node 上的 kubelet 进程实例化成一组相关的 Docker 容器并启动起来。在默认情况下，当 Pod 里的某个容器停止时，Kubernetes 会自动检测到这个问题并且重新启动这个 Pod (重启 Pod 里的所有容器)，如果 Pod 所在的 Node 宕机，则会将这个 Node 上的所有 Pod 重新调度到其他节点上。Pod、容器与 Node 的关系如图 1.7 所示。

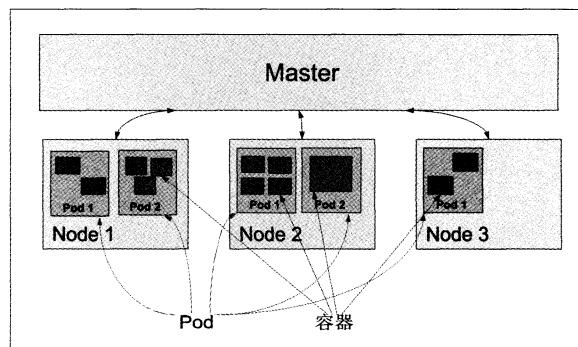


图 1.7 Pod、容器与 Node 的关系

Kubernetes 里的所有资源对象都可以采用 yaml 或者 JSON 格式的文件来定义或描述，下面是我们在之前 Hello World 例子里用到的 myweb 这个 Pod 的资源定义文件：

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: myweb
  labels:
    name: myweb
spec:
  containers:
  - name: myweb
    image: kubeguide/tomcat-app:v1
    ports:
    - containerPort: 8080
    env:
      - name: MYSQL_SERVICE_HOST
        value: 'mysql'
      - name: MYSQL_SERVICE_PORT
        value: '3306'

```

Kind 为 Pod 表明这是一个 Pod 的定义，metadata 里的 name 属性为 Pod 的名字，metadata 里还能定义资源对象的标签（Label），这里声明 myweb 拥有一个 name=myweb 的标签（Label）。Pod 里所包含的容器组的定义则在 spec 一节中声明，这里定义了一个名字为 myweb、对应镜像为 kubeguide/tomcat-app:v1 的容器，该容器注入了名为 MYSQL_SERVICE_HOST='mysql' 和 MYSQL_SERVICE_PORT='3306' 的环境变量（env 关键字），并且在 8080 端口（containerPort）上启动容器进程。Pod 的 IP 加上这里的容器端口（containerPort），就组成了一个新的概念——Endpoint，它代表着此 Pod 里的一个服务进程的对外通信地址。一个 Pod 也存在着具有多个 Endpoint 的情况，比如当我们把 Tomcat 定义为一个 Pod 的时候，可以对外暴露管理端口与服务端口这两个 Endpoint。

我们所熟悉的 Docker Volume 在 Kubernetes 里也有对应的概念——Pod Volume，后者有一些扩展，比如可以用分布式文件系统 GlusterFS 实现后端存储功能；Pod Volume 是定义在 Pod 之上，然后被各个容器挂载到自己的文件系统中的。

这里顺便提一下 Kubernetes 的 Event 概念，Event 是一个事件的记录，记录了事件的最早产生时间、最后重现时间、重复次数、发起者、类型，以及导致此事件的原因等众多信息。Event 通常会关联到某个具体的资源对象上，是排查故障的重要参考信息，之前我们看到 Node 的描述信息包括了 Event，而 Pod 同样有 Event 记录，当我们发现某个 Pod 迟迟无法创建时，可以用 kubectl describe pod xxxx 来查看它的描述信息，用来定位问题的原因，比如下面这个 Event 记录信息表明 Pod 里的一个容器被探针检测为失败一次：

Events:	FirstSeen	LastSeen	Count	From	SubobjectPath	Type	Reason
Message	-----	-----	-----	-----	-----	-----	-----
	10h	12m	32	{kubelet k8s-node-1} spec.containers{kube2sky}			

```
Warning  Unhealthy  Liveness probe failed: Get http://172.17.1.2:8080/healthz:  
net/http: request canceled (Client.Timeout exceeded while awaiting headers)
```

每个 Pod 都可以对其能使用的服务器上的计算资源设置限额，当前可以设置限额的计算资源有 CPU 与 Memory 两种，其中 CPU 的资源单位为 CPU (Core) 的数量，是一个绝对值而非相对值。

一个 CPU 的配额对于绝大多数容器来说是相当大的一个资源配额了，所以，在 Kubernetes 里，通常以千分之一的 CPU 配额为最小单位，用 m 来表示。通常一个容器的 CPU 配额被定义为 100~300m，即占用 0.1~0.3 个 CPU。由于 CPU 配额是一个绝对值，所以无论在拥有一个 Core 的机器上，还是在拥有 48 个 Core 的机器上，100m 这个配额所代表的 CPU 的使用量都是一样的。与 CPU 配额类似，Memory 配额也是一个绝对值，它的单位是内存字节数。

在 Kubernetes 里，一个计算资源进行配额限定需要设定以下两个参数。

- ◎ Requests：该资源的最小申请量，系统必须满足要求。
- ◎ Limits：该资源最大允许使用的量，不能被突破，当容器试图使用超过这个量的资源时，可能会被 Kubernetes Kill 并重启。

通常我们会把 Request 设置为一个比较小的数值，符合容器平时的工作负载情况下的资源需求，而把 Limit 设置为峰值负载情况下资源占用的最大量。比如下面这段定义，表明 MySQL 容器申请最少 0.25 个 CPU 及 64MiB 内存，在运行过程中 MySQL 容器所能使用的资源配额为 0.5 个 CPU 及 128MiB 内存：

```
spec:  
  containers:  
    - name: db  
      image: mysql  
      resources:  
        requests:  
          memory: "64Mi"  
          cpu: "250m"  
        limits:  
          memory: "128Mi"  
          cpu: "500m"
```

本节最后，笔者给出 Pod 及 Pod 周边对象的示意图作为总结，如图 1.8 所示，后面部分还会涉及这张图里的对象和概念，以进一步加强理解。

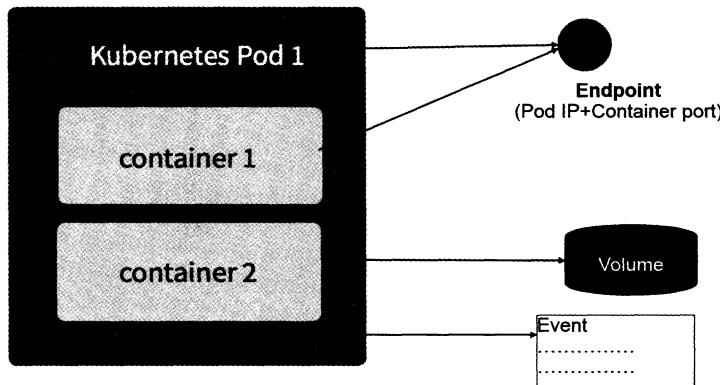


图 1.8 Pod 及周边对象

1.4.4 Label (标签)

Label 是 Kubernetes 系统中另外一个核心概念。一个 Label 是一个 `key=value` 的键值对，其中 `key` 与 `value` 由用户自己指定。Label 可以附加到各种资源对象上，例如 Node、Pod、Service、RC 等，一个资源对象可以定义任意数量的 Label，同一个 Label 也可以被添加到任意数量的资源对象上去，Label 通常在资源对象定义时确定，也可以在对象创建后动态添加或者删除。

我们可以通过给指定的资源对象捆绑一个或多个不同的 Label 来实现多维度的资源分组管理功能，以便于灵活、方便地进行资源分配、调度、配置、部署等管理工作。例如：部署不同版本的应用到不同的环境中；或者监控和分析应用（日志记录、监控、告警）等。一些常用的 Label 示例如下。

- ◎ 版本标签: "release": "stable", "release": "canary"...
- ◎ 环境标签: "environment": "dev", "environment": "qa", "environment": "production"
- ◎ 架构标签: "tier": "frontend", "tier": "backend", "tier": "middleware"
- ◎ 分区标签: "partition": "customerA", "partition": "customerB"...
- ◎ 质量管控标签: "track": "daily", "track": "weekly"

Label 相当于我们熟悉的“标签”，给某个资源对象定义一个 Label，就相当于给它打了一个标签，随后可以通过 Label Selector（标签选择器）查询和筛选拥有某些 Label 的资源对象，Kubernetes 通过这种方式实现了类似 SQL 的简单又通用的对象查询机制。

Label Selector 可以被类比为 SQL 语句中的 where 查询条件，例如，`name=redis-slave` 这个

Label Selector 作用于 Pod 时，可以被类比为 `select * from pod where pod's name = 'redis-slave'` 这样的语句。当前有两种 Label Selector 的表达式：基于等式的（Equality-based）和基于集合的（Set-based），前者采用“等式类”的表达式匹配标签，下面是一些具体的例子。

- ◎ `name = redis-slave`: 匹配所有具有标签 `name=redis-slave` 的资源对象。
- ◎ `env != production`: 匹配所有不具有标签 `env=production` 的资源对象，比如 `env=test` 就是满足此条件的标签之一。

而后者则使用集合操作的表达式匹配标签，下面是一些具体的例子。

- ◎ `name in (redis-master, redis-slave)`: 匹配所有具有标签 `name=redis-master` 或者 `name=redis-slave` 的资源对象。
- ◎ `name not in (php-frontend)`: 匹配所有不具有标签 `name=php-frontend` 的资源对象。

可以通过多个 Label Selector 表达式的组合实现复杂的条件选择，多个表达式之间用“，”进行分隔即可，几个条件之间是“AND”的关系，即同时满足多个条件，比如下面的例子：

```
name=redis-slave,env!=production  
name notin (php-frontend),env!=production
```

Label Selector 在 Kubernetes 中的重要使用场景有以下几处。

- ◎ `kube-controller` 进程通过资源对象 RC 上定义的 Label Selector 来筛选要监控的 Pod 副本的数量，从而实现 Pod 副本的数量始终符合预期设定的全自动控制流程。
- ◎ `kube-proxy` 进程通过 Service 的 Label Selector 来选择对应的 Pod，自动建立起每个 Service 到对应 Pod 的请求转发路由表，从而实现 Service 的智能负载均衡机制。
- ◎ 通过对某些 Node 定义特定的 Label，并且在 Pod 定义文件中使用 `NodeSelector` 这种标签调度策略，`kube-scheduler` 进程可以实现 Pod “定向调度”的特性。

在前面的留言板例子中，我们只使用了一个 `name=XXX` 的 Label Selector。让我们看一个更复杂的例子。假设为 Pod 定义了 3 个 Label： `release`、`env` 和 `role`，不同的 Pod 定义了不同的 Label 值，如图 1.9 所示，如果我们设置了“`role=frontend`”的 Label Selector，则会选取到 Node 1 和 Node 2 上的 Pod。

而设置“`release=beta`”的 Label Selector，则会选取到 Node 2 和 Node 3 上的 Pod，如图 1.10 所示。

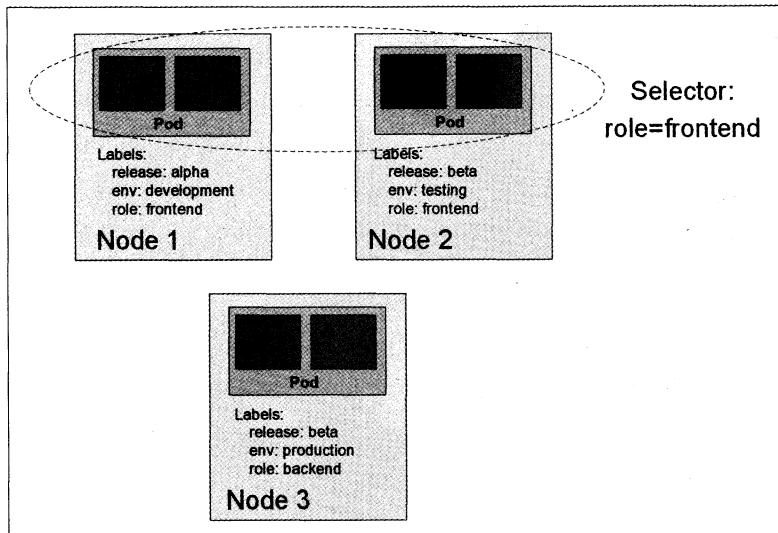


图 1.9 Label Selector 的作用范围 1

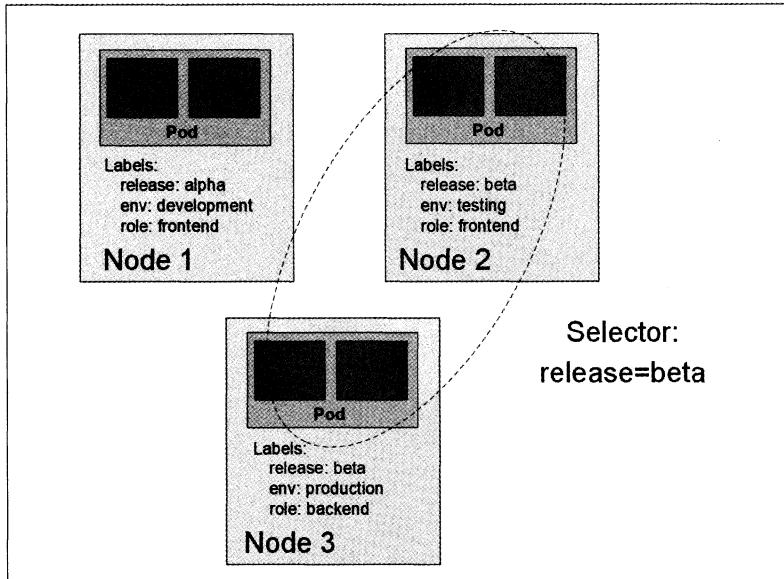


图 1.10 Label Selector 的作用范围 2

总结：使用 Label 可以给对象创建多组标签，Label 和 Label Selector 共同构成了 Kubernetes 系统中最核心的应用模型，使得被管理对象能够被精细地分组管理，同时实现了整个集群的高可用性。

1.4.5 Replication Controller (RC)

之前已经对 Replication Controller (以后简称 RC) 的定义和作用做了一些说明，本节对 RC 的概念进行深入描述。

RC 是 Kubernetes 系统中的核心概念之一，简单来说，它其实是定义了一个期望的场景，即声明某种 Pod 的副本数量在任意时刻都符合某个预期值，所以 RC 的定义包括如下几个部分。

- ◎ Pod 期待的副本数 (replicas)。
- ◎ 用于筛选目标 Pod 的 Label Selector。
- ◎ 当 Pod 的副本数量小于预期数量的时候，用于创建新 Pod 的 Pod 模板 (template)。

下面是一个完整的 RC 定义的例子，即确保拥有 tier=frontend 标签的这个 Pod (运行 Tomcat 容器) 在整个 Kubernetes 集群中始终只有一个副本。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    tier: frontend
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
        - name: tomcat-demo
          image: tomcat
          imagePullPolicy: IfNotPresent
          env:
            - name: GET_HOSTS_FROM
              value: dns
          ports:
            - containerPort: 80
```

当我们定义了一个 RC 并提交到 Kubernetes 集群中以后，Master 节点上的 Controller Manager 组件就得到通知，定期巡检系统中当前存活的目标 Pod，并确保目标 Pod 实例的数量刚好等于此 RC 的期望值，如果有过多的 Pod 副本在运行，系统就会停掉一些 Pod，否则系统就会再自动创建一些 Pod。可以说，通过 RC，Kubernetes 实现了用户应用集群的高可用性，并且大大减少了系统管理员在传统 IT 环境中需要完成的许多手工运维工作（如主机监控脚本、应用监控脚

本、故障恢复脚本等)。

下面我们以3个Node节点的集群为例,说明Kubernetes如何通过RC来实现Pod副本数量自动控制的机制。假如我们的RC里定义redis-slave这个Pod需要保持3个副本,系统将可能在其中的两个Node上创建Pod。图1.11描述了在两个Node上创建redis-slave Pod的情形。

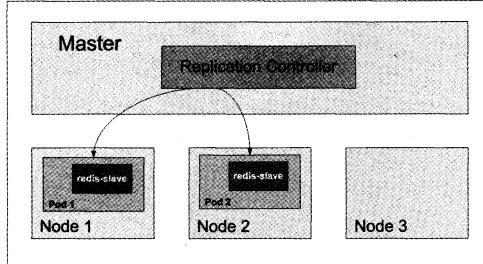


图1.11 在两个Node上创建redis-slave Pod

假设Node 2上的Pod 2意外终止,根据RC定义的replicas数量2,Kubernetes将会自动创建并启动一个新的Pod,以保证整个集群中始终有两个redis-slave Pod在运行。

如图1.12所示,系统可能选择Node 3或者Node 1来创建一个新的Pod。

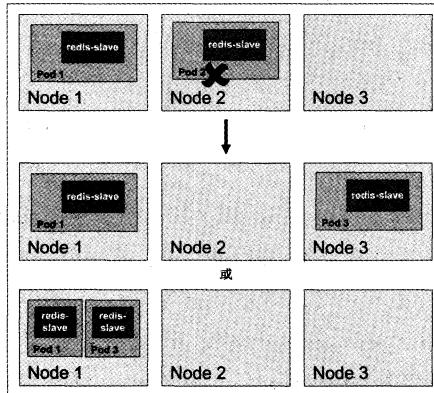


图1.12 根据RC定义创建新的Pod

此外,在运行时,我们可以通过修改RC的副本数量,来实现Pod的动态缩放(Scaling)功能,这可以通过执行kubectl scale命令来一键完成:

```
$ kubectl scale rc redis-slave --replicas=3
scaled
```

Scaling的执行结果如图1.13所示。

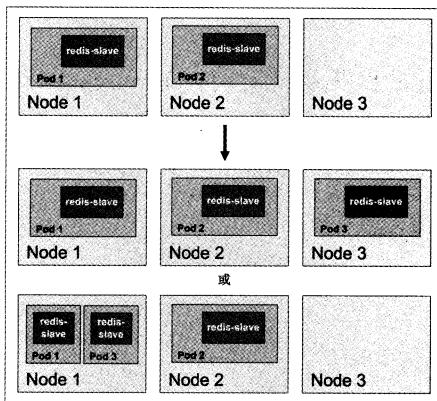


图 1.13 Scaling 的执行结果

需要注意的是，删除 RC 并不会影响通过该 RC 已创建好的 Pod。为了删除所有 Pod，可以设置 replicas 的值为 0，然后更新该 RC。另外，`kubectl` 提供了 `stop` 和 `delete` 命令来一次性删除 RC 和 RC 控制的全部 Pod。

当我们的应用升级时，通常会通过 Build 一个新的 Docker 镜像，并用新的镜像版本来替代旧的版本的方式达到目的。在系统升级的过程中，我们希望是平滑的方式，比如当前系统中 10 个对应的旧版本的 Pod，最佳的方式是旧版本的 Pod 每次停止一个，同时创建一个新版本的 Pod，在整个升级过程中，此消彼长，而运行中的 Pod 数量始终是 10 个，几分钟以后，当所有的 Pod 都已经是新版本的时候，升级过程完成。通过 RC 的机制，Kubernetes 很容易就实现了这种高级实用的特性，被称为“滚动升级”（Rolling Update），具体的操作方法详见第 4 章。

由于 Replication Controller 与 Kubernetes 代码中的模块 Replication Controller 同名，同时这个词也无法准确表达它的本意，所以在 Kubernetes 1.2 的时候，它就升级成了另外一个新的概念——Replica Set，官方解释为“下一代的 RC”，它与 RC 当前存在的唯一区别是：Replica Sets 支持基于集合的 Label selector（Set-based selector），而 RC 只支持基于等式的 Label Selector（equality-based selector），这使得 Replica Set 的功能更强，下面是等价于之前 RC 例子的 Replica Set 的定义（省去了 Pod 模板部分的内容）：

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
```

```
template:  
.....
```

kubectl 命令行工具适用于 RC 的绝大部分命令都同样适用于 Replica Set。此外，当前我们很少单独使用 Replica Set，它主要被 Deployment 这个更高层的资源对象所使用，从而形成一整套 Pod 创建、删除、更新的编排机制。当我们使用 Deployment 时，无须关心它是如何创建和维护 Replica Set 的，这一切都是自动发生的。

Replica Set 与 Deployment 这两个重要资源对象逐步替换了之前的 RC 的作用，是 Kubernetes 1.3 里 Pod 自动扩容（伸缩）这个告警功能实现的基础，也将继续在 Kubernetes 未来的版本中发挥重要的作用。

最后我们总结一下关于 RC（Replica Set）的一些特性与作用。

- ◎ 在大多数情况下，我们通过定义一个 RC 实现 Pod 的创建过程及副本数量的自动控制。
- ◎ RC 里包括完整的 Pod 定义模板。
- ◎ RC 通过 Label Selector 机制实现对 Pod 副本的自动控制。
- ◎ 通过改变 RC 里的 Pod 副本数量，可以实现 Pod 的扩容或缩容功能。
- ◎ 通过改变 RC 里 Pod 模板中的镜像版本，可以实现 Pod 的滚动升级功能。

1.4.6 Deployment

Deployment 是 Kubernetes 1.2 引入的新概念，引入的目的是为了更好地解决 Pod 的编排问题。为此，Deployment 在内部使用了 Replica Set 来实现目的，无论从 Deployment 的作用与目的、它的 YAML 定义，还是从它的具体命令行操作来看，我们都可以把它看作 RC 的一次升级，两者的相似度超过 90%。

Deployment 相对于 RC 的一个最大升级是我们可以随时知道当前 Pod “部署”的进度。实际上由于一个 Pod 的创建、调度、绑定节点及在目标 Node 上启动对应的容器这一完整过程需要一定的时间，所以我们期待系统启动 N 个 Pod 副本的目标状态，实际上是一个连续变化的“部署过程”导致的最终状态。

Deployment 的典型使用场景有以下几个。

- ◎ 创建一个 Deployment 对象来生成对应的 Replica Set 并完成 Pod 副本的创建过程。
- ◎ 检查 Deployment 的状态来看部署动作是否完成（Pod 副本的数量是否达到预期的值）。
- ◎ 更新 Deployment 以创建新的 Pod（比如镜像升级）。
- ◎ 如果当前 Deployment 不稳定，则回滚到一个早先的 Deployment 版本。

◎ 挂起或者恢复一个 Deployment。

Deployment 的定义与 Replica Set 的定义很类似，除了 API 声明与 Kind 类型等有所区别：

```
apiVersion: extensions/v1beta1           apiVersion: v1
kind: Deployment                          kind: ReplicaSet
metadata:                                 metadata:
  name: nginx-deployment                name: nginx-repset
```

下面我们通过运行一些例子来一起直观地感受这个新概念。首先创建一个名为 tomcat-deployment.yaml 的 Deployment 描述文件，内容如下：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: app-demo
        tier: frontend
    spec:
      containers:
        - name: tomcat-demo
          image: tomcat
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
```

运行下述命令创建 Deployment：

```
# kubectl create -f tomcat-deployment.yaml
deployment "tomcat-deploy" created
```

运行下述命令查看 Deployment 的信息：

```
# kubectl get deployments
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
tomcat-deploy  1         1         1           1           4m
```

对上述输出中涉及的数量解释如下。

- ◎ DESIRED：Pod 副本数量的期望值，即 Deployment 里定义的 Replica。

- ◎ CURRENT: 当前 Replica 的值, 实际上是 Deployment 所创建的 Replica Set 里的 Replica 值, 这个值不断增加, 直到达到 DESIRED 为止, 表明整个部署过程完成。
- ◎ UP-TO-DATE: 最新版本的 Pod 的副本数量, 用于指示在滚动升级的过程中, 有多少个 Pod 副本已经成功升级。
- ◎ AVAILABLE: 当前集群中可用的 Pod 副本数量, 即集群中当前存活的 Pod 数量。

运行下述命令查看对应的 Replica Set, 我们看到它的命名跟 Deployment 的名字有关系:

```
# kubectl get rs
NAME           DESIRED   CURRENT   AGE
tomcat-deploy-1640611518   1         1         1m
```

运行下述命令查看创建的 Pod, 我们发现 Pod 的命名以 Deployment 对应的 Replica Set 的名字为前缀, 这种命名很清晰地表明了一个 Replica Set 创建了哪些 Pod, 对于 Pod 滚动升级这种复杂的过程来说, 很容易排查错误:

```
# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
tomcat-deploy-1640611518-zhrsc   1/1     Running   0          3m
```

运行 `kubectl describe deployments`, 可以清楚地看到 Deployment 控制的 Pod 的水平扩展过程, 此命令的输出比较多, 这里不再赘述。

1.4.7 Horizontal Pod Autoscaler (HPA)

在前两节提到过, 通过手工执行 `kubectl scale` 命令, 我们可以实现 Pod 扩容或缩容。如果仅仅到此为止, 显然不符合谷歌对 Kubernetes 的定位目标——自动化、智能化。在谷歌看来, 分布式系统要能够根据当前负载的变化情况自动触发水平扩展或缩容的行为, 因为这一过程可能是频繁发生的、不可预料的, 所以手动控制的方式是不现实的。

因此, Kubernetes 的 1.0 版本实现后, 这帮大牛们就已经在默默研究 Pod 智能扩容的特性了, 并在 Kubernetes 1.1 的版本中首次发布这一重量级新特性——Horizontal Pod Autoscaling。随后的 1.2 版本中 HPA 被升级为稳定版本 (`apiVersion: autoscaling/v1`), 但同时仍然保留旧版本 (`apiVersion: extensions/v1beta1`), 官方的计划是在 1.3 版本里先移除旧版本, 并且会在 1.4 版本里彻底移除旧版本的支持。

Horizontal Pod Autoscaling 简称 HPA, 意思是 Pod 横向自动扩容, 与之前的 RC、Deployment 一样, 也属于一种 Kubernetes 资源对象。通过追踪分析 RC 控制的所有目标 Pod 的负载变化情况, 来确定是否需要针对性地调整目标 Pod 的副本数, 这是 HPA 的实现原理。当前, HPA 可以有以下两种方式作为 Pod 负载的度量指标。

- ◎ CPUUtilizationPercentage。

- ◎ 应用程序自定义的度量指标，比如服务在每秒内的相应的请求数（TPS 或 QPS）。

CPUUtilizationPercentage 是一个算术平均值，即目标 Pod 所有副本自身的 CPU 利用率的平均值。一个 Pod 自身的 CPU 利用率是该 Pod 当前 CPU 的使用量除以它的 Pod Request 的值，比如我们定义一个 Pod 的 Pod Request 为 0.4，而当前 Pod 的 CPU 使用量为 0.2，则它的 CPU 使用率为 50%，如此一来，我们就可以就算出来一个 RC 控制的所有 Pod 副本的 CPU 利用率的算术平均值了。如果某一时刻 CPUUtilizationPercentage 的值超过 80%，则意味着当前的 Pod 副本数很可能不足以支撑接下来更多的请求，需要进行动态扩容，而当请求高峰时段过去后，Pod 的 CPU 利用率又会降下来，此时对应的 Pod 副本数应该自动减少到一个合理的水平。

CPUUtilizationPercentage 计算过程中使用到的 Pod 的 CPU 使用量通常是 1 分钟内的平均值，目前通过查询 Heapster 扩展组件来得到这个值，所以需要安装部署 Heapster，这样一来便增加了系统的复杂度和实施 HPA 特性的复杂度，因此，未来的计划是 Kubernetes 自身实现一个基础性能数据采集模块，从而更好地支持 HPA 和其他需要用到基础性能数据的功能模块。此外，我们也看到，如果目标 Pod 没有定义 Pod Request 的值，则无法使用 CPUUtilizationPercentage 来实现 Pod 横向自动扩容的能力。除了使用 CPUUtilizationPercentage，Kubernetes 从 1.2 版本开始，尝试支持应用程序自定义的度量指标，目前仍然为实验特性，不建议在生产环境中使用。

下面是 HPA 定义的一个具体例子：

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    kind: Deployment
    name: php-apache
  targetCPUUtilizationPercentage: 90
```

根据上面的定义，我们可以知道这个 HPA 控制的目标对象为一个名叫 php-apache 的 Deployment 里的 Pod 副本，当这些 Pod 副本的 CPUUtilizationPercentage 的值超过 90% 时会触发自动动态扩容行为，扩容或缩容时必须满足的一个约束条件是 Pod 的副本数要介于 1 与 10 之间。

除了可以通过直接定义 yaml 文件并且调用 kubectl create 的命令来创建一个 HPA 资源对象的方式，我们还能通过下面的简单命令行直接创建等价的 HPA 对象：

```
# kubectl autoscale deployment php-apache --cpu-percent=90 --min=1 --max=10
```

第2章将会给出一个完整的HPA例子来说明其用法和功能。

1.4.8 Service（服务）

1. 概述

Service也是Kubernetes里的最核心的资源对象之一，Kubernetes里的每个Service其实是我们经常提起的微服务架构中的一个“微服务”，之前我们所说的Pod、RC等资源对象其实都是为这节所说的“服务”——Kubernetes Service做“嫁衣”的。图1.14显示了Pod、RC与Service的逻辑关系。

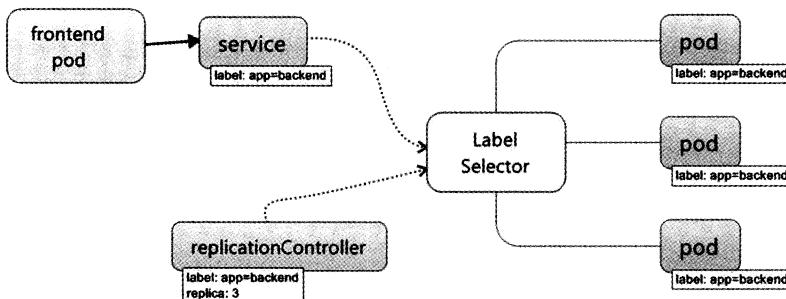


图1.14 Pod、RC与Service的关系

从图1.14中我们看到，Kubernetes的Service定义了一个服务的访问入口地址，前端的应用(Pod)通过这个入口地址访问其背后的一组由Pod副本组成的集群实例，Service与其后端Pod副本集群之间则是通过Label Selector来实现“无缝对接”的。而RC的作用实际上是保证Service的服务能力和服务质量始终处于预期的标准。

通过分析、识别并建模系统中的所有服务为微服务——Kubernetes Service，最终我们的系统由多个提供不同业务能力而又彼此独立的微服务单元所组成，服务之间通过TCP/IP进行通信，从而形成了我们强大而又灵活的弹性网格，拥有了强大的分布式能力、弹性扩展能力、容错能力，与此同时，我们的程序架构也变得简单和直观许多，如图1.15所示。

既然每个Pod都会被分配一个单独的IP地址，而且每个Pod都提供了一个独立的Endpoint(Pod IP+ContainerPort)以被客户端访问，现在多个Pod副本组成了一个集群来提供服务，那么客户端如何来访问它们呢？一般的做法是部署一个负载均衡器(软件或硬件)，为这组Pod开启一个对外的服务端口如8000端口，并且将这些Pod的Endpoint列表加入8000端口的转发列表中，客户端就可以通过负载均衡器的对外IP地址+服务端口来访问此服务，而客户端的请求最后会被转发到哪个Pod，则由负载均衡器的算法所决定。

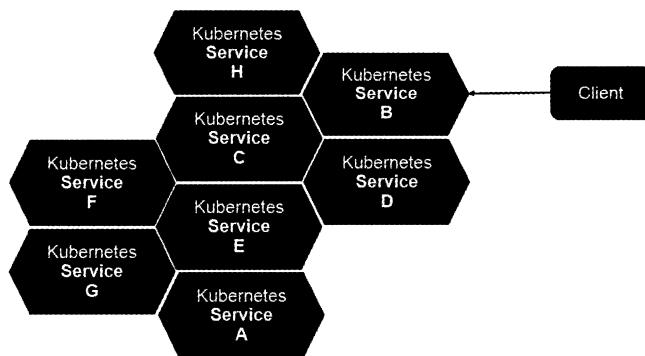


图 1.15 Kubernetes 所提供的微服务网格架构

Kubernetes 也遵循了上述常规做法，运行在每个 Node 上的 kube-proxy 进程其实就是一个智能的软件负载均衡器，它负责把对 Service 的请求转发到后端的某个 Pod 实例上，并在内部实现服务的负载均衡与会话保持机制。但 Kubernetes 发明了一种很巧妙又影响深远的设计：Service 不是共用一个负载均衡器的 IP 地址，而是每个 Service 分配了一个全局唯一的虚拟 IP 地址，这个虚拟 IP 被称为 Cluster IP。这样一来，每个服务就变成了具备唯一 IP 地址的“通信节点”，服务调用就变成了最基础的 TCP 网络通信问题。

我们知道，Pod 的 Endpoint 地址会随着 Pod 的销毁和重新创建而发生改变，因为新 Pod 的 IP 地址与之前旧 Pod 的不同。而 Service 一旦创建，Kubernetes 就会自动为它分配一个可用的 Cluster IP，而且在 Service 的整个生命周期内，它的 Cluster IP 不会发生改变。于是，服务发现这个棘手的问题在 Kubernetes 的架构里也得以轻松解决：只要用 Service 的 Name 与 Service 的 Cluster IP 地址做一个 DNS 域名映射即可完美解决问题。现在想想，这真是一个很棒的设计。

说了这么久，下面我们动手创建一个 Service，来加深对它的理解。首先我们创建一个名为 tomcat-service.yaml 的定义文件，内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
  - port: 8080
  selector:
    tier: frontend
```

上述内容定义了一个名为“tomcat-service”的 Service，它的服务端口为 8080，拥有“tier = frontend”这个 Label 的所有 Pod 实例都属于它，运行下面的命令进行创建：

```
#kubectl create -f tomcat-server.yaml
service "tomcat-service" created
```

注意到我们之前在 `tomcat-deployment.yaml` 里定义的 Tomcat 的 Pod 刚好拥有这个标签，所以我们刚才创建的 `tomcat-service` 已经对应到了一个 Pod 实例，运行下面的命令可以查看 `tomcat-service` 的 Endpoint 列表，其中 172.17.1.3 是 Pod 的 IP 地址，端口 8080 是 Container 暴露的端口：

```
# kubectl get endpoints
NAME           ENDPOINTS          AGE
kubernetes     192.168.18.131:6443   15d
tomcat-service 172.17.1.3:8080      1m
```

你可能有疑问：“说好的 Service 的 Cluster IP 呢？怎么没有看到？”我们运行下面的命令即可看到 `tomcat-service` 被分配的 Cluster IP 及更多的信息：

```
# kubectl get svc tomcat-service -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2016-07-21T17:05:52Z
  name: tomcat-service
  namespace: default
  resourceVersion: "23964"
  selfLink: /api/v1/namespaces/default/services/tomcat-service
  uid: 61987d3c-4f65-11e6-a9d8-000c29ed42c1
spec:
  clusterIP: 169.169.65.227
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    tier: frontend
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

在 `spec.ports` 的定义中，`targetPort` 属性用来确定提供该服务的容器所暴露（EXPOSE）的端口号，即具体业务进程在容器内的 `targetPort` 上提供 TCP/IP 接入；而 `port` 属性则定义了 Service 的虚端口。前面我们定义 Tomcat 服务的时候，没有指定 `targetPort`，则默认 `targetPort` 与 `port` 相同。

接下来，我们来看看 Service 的多端口问题。

很多服务都存在多个端口的问题，通常一个端口提供业务服务，另外一个端口提供管理服务，比如 Mycat、Codis 等常见中间件。Kubernetes Service 支持多个 Endpoint，在存在多个 Endpoint 的情况下，要求每个 Endpoint 定义一个名字来区分。下面是 Tomcat 多端口的 Service 定义样例：

```
apiVersion: v1
kind: Service
```

```
metadata:  
  name: tomcat-service  
spec:  
  ports:  
    - port: 8080  
      name: service-port  
    - port: 8005  
      name: shutdown-port  
  selector:  
    tier: frontend
```

多端口为什么需要给每个端口命名呢？这就涉及 Kubernetes 的服务发现机制了，我们接下来进行讲解。

2. Kubernetes 的服务发现机制

任何分布式系统都会涉及“服务发现”这个基础问题，大部分分布式系统通过提供特定的 API 接口来实现服务发现的功能，但这样做会导致平台的侵入性比较强，也增加了开发测试的困难。Kubernetes 则采用了直观朴素的思路去解决这个棘手的问题。

首先，每个 Kubernetes 中的 Service 都有一个唯一的 Cluster IP 以及唯一的名字，而名字是由开发者自己定义的，部署的时候也没必要改变，所以完全可以固定在配置中。接下来的问题就是如何通过 Service 的名字找到对应的 Cluster IP？

最早的时候 Kubernetes 采用了 Linux 环境变量的方式解决这个问题，即每个 Service 生成一些对应的 Linux 环境变量 (ENV)，并在每个 Pod 的容器在启动时，自动注入这些环境变量，以下是从 tomcat-service 产生的环境变量条目：

```
TOMCAT_SERVICE_SERVICE_HOST=169.169.41.218  
TOMCAT_SERVICE_SERVICE_PORT_SERVICE_PORT=8080  
TOMCAT_SERVICE_SERVICE_PORT_SHUTDOWN_PORT=8005  
TOMCAT_SERVICE_SERVICE_PORT=8080  
TOMCAT_SERVICE_PORT_8005_TCP_PORT=8005  
TOMCAT_SERVICE_PORT=tcp://169.169.41.218:8080  
TOMCAT_SERVICE_PORT_8080_TCP_ADDR=169.169.41.218  
TOMCAT_SERVICE_PORT_8080_TCP=tcp://169.169.41.218:8080  
TOMCAT_SERVICE_PORT_8080_TCP_PROTO=tcp  
TOMCAT_SERVICE_PORT_8080_TCP_PORT=8080  
TOMCAT_SERVICE_PORT_8005_TCP=tcp://169.169.41.218:8005  
TOMCAT_SERVICE_PORT_8005_TCP_ADDR=169.169.41.218  
TOMCAT_SERVICE_PORT_8005_TCP_PROTO=tcp
```

上述环境变量中，比较重要的是前 3 条环境变量，我们可以看到，每个 Service 的 IP 地址及端口都是有标准的命名规范的，遵循这个命名规范，就可以通过代码访问系统环境变量的方式得到所需的信息，实现服务调用。

考虑到环境变量的方式获取 Service 的 IP 与端口的方式仍然不太方便，不够直观，后来 Kubernetes 通过 Add-On 增值包的方式引入了 DNS 系统，把服务名作为 DNS 域名，这样一来，程序就可以直接使用服务名来建立通信连接了。目前 Kubernetes 上的大部分应用都已经采用了 DNS 这些新兴的服务发现机制，后面的章节中我们会讲述如何部署这套 DNS 系统。

3. 外部系统访问 Service 的问题

为了更加深入地理解和掌握 Kubernetes，我们需要弄明白 Kubernetes 里的“三种 IP”这个关键问题，这三种 IP 分别如下。

- ◎ Node IP：Node 节点的 IP 地址。
- ◎ Pod IP：Pod 的 IP 地址。
- ◎ Cluster IP：Service 的 IP 地址。

首先，Node IP 是 Kubernetes 集群中每个节点的物理网卡的 IP 地址，这是一个真实存在的物理网络，所有属于这个网络的服务器之间都能通过这个网络直接通信，不管它们中是否有部分节点不属于这个 Kubernetes 集群。这也表明了 Kubernetes 集群之外的节点访问 Kubernetes 集群之内的某个节点或者 TCP/IP 服务的时候，必须要通过 Node IP 进行通信。

其次，Pod IP 是每个 Pod 的 IP 地址，它是 Docker Engine 根据 docker0 网桥的 IP 地址段进行分配的，通常是一个虚拟的二层网络，前面我们说过，Kubernetes 要求位于不同 Node 上的 Pod 能够彼此直接通信，所以 Kubernetes 里一个 Pod 里的容器访问另外一个 Pod 里的容器，就是通过 Pod IP 所在的虚拟二层网络进行通信的，而真实的 TCP/IP 流量则是通过 Node IP 所在的物理网卡流出的。

最后，我们说说 Service 的 Cluster IP，它也是一个虚拟的 IP，但更像是一个“伪造”的 IP 网络，原因有以下几点。

- ◎ Cluster IP 仅仅作用于 Kubernetes Service 这个对象，并由 Kubernetes 管理和分配 IP 地址（来源于 Cluster IP 地址池）。
- ◎ Cluster IP 无法被 Ping，因为没有一个“实体网络对象”来响应。
- ◎ Cluster IP 只能结合 Service Port 组成一个具体的通信端口，单独的 Cluster IP 不具备 TCP/IP 通信的基础，并且它们属于 Kubernetes 集群这样一个封闭的空间，集群之外的节点如果要访问这个通信端口，则需要做一些额外的工作。
- ◎ 在 Kubernetes 集群之内，Node IP 网、Pod IP 网与 Cluster IP 网之间的通信，采用的是 Kubernetes 自己设计的一种编程方式的特殊的路由规则，与我们所熟知的 IP 路由有很大的不同。

根据上面的分析和总结，我们基本明白了：Service 的 Cluster IP 属于 Kubernetes 集群内部的地址，无法在集群外部直接使用这个地址。那么矛盾来了：实际上我们开发的业务系统中肯定多少有一部分服务是要提供给 Kubernetes 集群外部的应用或者用户来使用的，典型的例子就是 Web 端的服务模块，比如上面的 tomcat-service，那么用户怎么访问它？

采用 NodePort 是解决上述问题的最直接、最有效、最常用的做法。具体做法如下，以 tomcat-service 为例，我们在 Service 的定义里做如下扩展即可（黑体字部分）：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 31002
  selector:
    tier: frontend
```

其中，nodePort:31002 这个属性表明我们手动指定 tomcat-service 的 NodePort 为 31002，否则 Kubernetes 会自动分配一个可用的端口。接下来，我们在浏览器里访问 `http://<nodePort IP>:31002/`，就可以看到 Tomcat 的欢迎界面了，如图 1.16 所示。

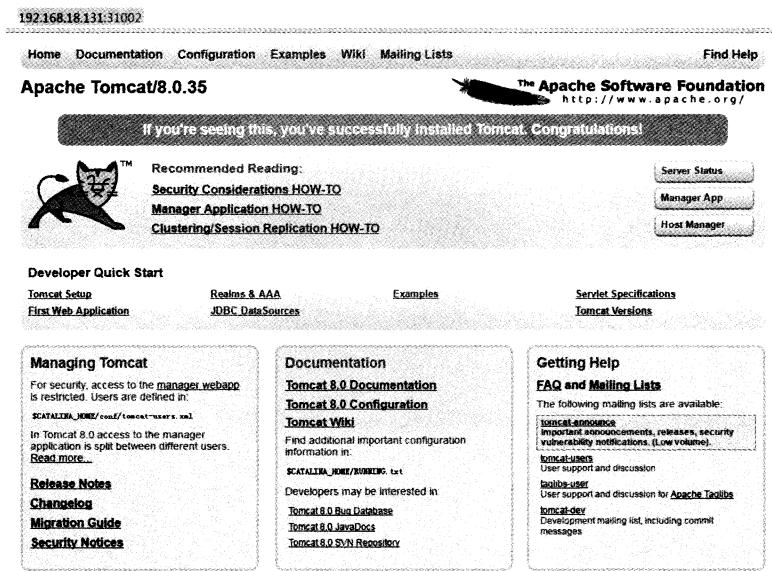


图 1.16 通过 NodePort 访问 Service

NodePort的实现方式是在Kubernetes集群里的每个Node上为需要外部访问的Service开启一个对应的TCP监听端口，外部系统只要用任意一个Node的IP地址+具体的NodePort端口号即可访问此服务，在任意Node上运行netstat命令，我们就可以看到有NodePort端口被监听：

```
# netstat -tlp | grep 31002
tcp6 0 0 [::]:31002 [::]:* LISTEN 1125/kube-proxy
```

但NodePort还没有完全解决外部访问Service的所有问题，比如负载均衡问题，假如我们的集群中有10个Node，则此时最好有一个负载均衡器，外部的请求只需访问此负载均衡器的IP地址，由负载均衡器负责转发流量到后面某个Node的NodePort上。如图1.17所示。

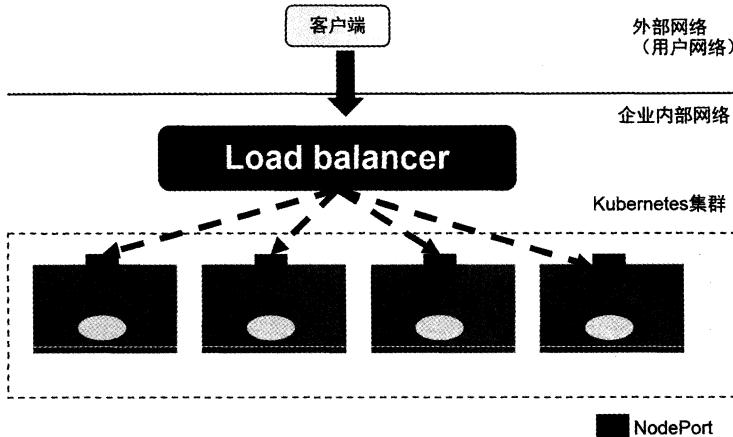


图1.17 NodePort与Load balancer

图1.17中的Load balancer组件独立于Kubernetes集群之外，通常是一个硬件的负载均衡器，或者是以软件方式实现的，例如HAProxy或者Nginx。对于每个Service，我们通常需要配置一个对应的Load balancer实例来转发流量到后端的Node上，这的确增加了工作量及出错的概率。于是Kubernetes提供了自动化的解决方案，如果我们的集群运行在谷歌的GCE公有云上，那么只要我们把Service的type=NodePort改为type=LoadBalancer，此时Kubernetes会自动创建一个对应的Load balancer实例并返回它的IP地址供外部客户端使用。其他公有云提供商只要实现了支持此特性的驱动，则也可以达到上述目的。此外，裸机上的类似机制(Bare Metal Service Load Balancers)也正在被开发。

1.4.9 Volume（存储卷）

Volume是Pod中能够被多个容器访问的共享目录。Kubernetes的Volume概念、用途和目的与Docker的Volume比较类似，但两者不能等价。首先，Kubernetes中的Volume定义在Pod

上，然后被一个 Pod 里的多个容器挂载到具体的文件目录下；其次，Kubernetes 中的 Volume 与 Pod 的生命周期相同，但与容器的生命周期不相关，当容器终止或者重启时，Volume 中的数据也不会丢失。最后，Kubernetes 支持多种类型的 Volume，例如 GlusterFS、Ceph 等先进的分布式文件系统。

Volume 的使用也比较简单，在大多数情况下，我们先在 Pod 上声明一个 Volume，然后在容器里引用该 Volume 并 Mount 到容器里的某个目录上。举例来说，我们要给之前的 Tomcat Pod 增加一个名字为 dataVol 的 Volume，并且 Mount 到容器的 /mydata-data 目录上，则只要对 Pod 的定义文件做如下修正即可（注意黑体字部分）：

```
template:
  metadata:
    labels:
      app: app-demo
      tier: frontend
  spec:
    volumes:
      - name: datavol
        emptyDir: {}
    containers:
      - name: tomcat-demo
        image: tomcat
        volumeMounts:
          - mountPath: /mydata-data
            name: datavol
        imagePullPolicy: IfNotPresent
```

除了可以让一个 Pod 里的多个容器共享文件、让容器的数据写到宿主机的磁盘上或者写文件到网络存储中，Kubernetes 的 Volume 还扩展出了一种非常有实用价值的功能，即容器配置文件集中化定义与管理，这是通过 ConfigMap 这个新的资源对象来实现的，后面我们会详细说明。

Kubernetes 提供了非常丰富的 Volume 类型，下面逐一进行说明。

1. emptyDir

一个 emptyDir Volume 是在 Pod 分配到 Node 时创建的。从它的名称就可以看出，它的初始内容为空，并且无须指定宿主机上对应的目录文件，因为这是 Kubernetes 自动分配的一个目录，当 Pod 从 Node 上移除时，emptyDir 中的数据也会被永久删除。emptyDir 的一些用途如下。

- ◎ 临时空间，例如用于某些应用程序运行时所需的临时目录，且无须永久保留。
- ◎ 长时间任务的中间过程 CheckPoint 的临时保存目录。
- ◎ 一个容器需要从另一个容器中获取数据的目录（多容器共享目录）。

目前，用户无法控制 emptyDir 使用的介质种类。如果 kubelet 的配置是使用硬盘，那么所有 emptyDir 都将创建在该硬盘上。Pod 在将来可以设置 emptyDir 是位于硬盘、固态硬盘上还是基于内存的 tmpfs 上，上面的例子便采用了 emptyDir 类的 Volume。

2. hostPath

hostPath 为在 Pod 上挂载宿主机上的文件或目录，它通常可以用于以下几方面。

- ◎ 容器应用程序生成的日志文件需要永久保存时，可以使用宿主机的高速文件系统进行存储。
- ◎ 需要访问宿主机上 Docker 引擎内部数据结构的容器应用时，可以通过定义 hostPath 为宿主机 /var/lib/docker 目录，使容器内部应用可以直接访问 Docker 的文件系统。

在使用这种类型的 Volume 时，需要注意以下几点。

- ◎ 在不同的 Node 上具有相同配置的 Pod 可能会因为宿主机上的目录和文件不同而导致对 Volume 上目录和文件的访问结果不一致。
- ◎ 如果使用了资源配额管理，则 Kubernetes 无法将 hostPath 在宿主机上使用的资源纳入管理。

在下面的例子中使用宿主机的 /data 目录定义了一个 hostPath 类型的 Volume：

```
volumes:
- name: "persistent-storage"
  hostPath:
    path: "/data"
```

3. gcePersistentDisk

使用这种类型的 Volume 表示使用谷歌公有云提供的永久磁盘（Persistent Disk，PD）存放 Volume 的数据，它与 EmptyDir 不同，PD 上的内容会被永久保存，当 Pod 被删除时，PD 只是被卸载（Unmount），但不会被删除。需要注意的是，你需要先创建一个永久磁盘（PD），才能使用 gcePersistentDisk。

使用 gcePersistentDisk 有以下一些限制条件。

- ◎ Node（运行 kubelet 的节点）需要是 GCE 虚拟机。
- ◎ 这些虚拟机需要与 PD 存在于相同的 GCE 项目和 Zone 中。

通过 gcloud 命令即可创建一个 PD：

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

定义 gcePersistentDisk 类型的 Volume 的示例如下：

```
volumes:
- name: test-volume
  # This GCE PD must already exist.
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

4. awsElasticBlockStore

与 GCE 类似，该类型的 Volume 使用亚马逊公有云提供的 EBS Volume 存储数据，需要先创建一个 EBS Volume 才能使用 awsElasticBlockStore。

使用 awsElasticBlockStore 的一些限制条件如下。

- ◎ Node（运行 kubelet 的节点）需要是 AWS EC2 实例。
- ◎ 这些 AWS EC2 实例需要与 EBS volume 存在于相同的 region 和 availability-zone 中。
- ◎ EBS 只支持单个 EC2 实例 mount 一个 volume。

通过 aws ec2 create-volume 命令可以创建一个 EBS volume：

```
aws ec2 create-volume --availability-zone eu-west-1a --size 10 --volume-type gp2
```

定义 awsElasticBlockStore 类型的 Volume 的示例如下：

```
volumes:
- name: test-volume
  # This AWS EBS volume must already exist.
  awsElasticBlockStore:
    volumeID: aws://<availability-zone>/<volume-id>
    fsType: ext4
```

5. NFS

使用 NFS 网络文件系统提供的共享目录存储数据时，我们需要在系统中部署一个 NFS Server。定义 NFS 类型的 Volume 的示例如下：

```
volumes:
- name: nfs
  nfs:
    # 改为你的 NFS 服务器地址
    server: nfs-server.localhost
    path: "/"
```

6. 其他类型的 Volume

- ◎ iscsi：使用 iSCSI 存储设备上的目录挂载到 Pod 中。

- ◎ flocker: 使用 Flocker 来管理存储卷。
- ◎ glusterfs: 使用开源 GlusterFS 网络文件系统的目录挂载到 Pod 中。
- ◎ rbd: 使用 Linux 块设备共享存储 (Rados Block Device) 挂载到 Pod 中。
- ◎ gitRepo: 通过挂载一个空目录，并从 GIT 库 clone 一个 git repository 以供 Pod 使用。
- ◎ secret: 一个 secret volume 用于为 Pod 提供加密的信息，你可以将定义在 Kubernetes 中的 secret 直接挂载为文件让 Pod 访问。secret volume 是通过 tmpfs (内存文件系统) 实现的，所以这种类型的 volume 总是不会持久化的。

1.4.10 Persistent Volume

之前我们提到的 Volume 是定义在 Pod 上的，属于“计算资源”的一部分，而实际上，“网络存储”是相对独立于“计算资源”而存在的一种实体资源。比如在使用虚机的情况下，我们通常会先定义一个网络存储，然后从中划出一个“网盘”并挂接到虚机上。Persistent Volume (简称 PV) 和与之相关联的 Persistent Volume Claim (简称 PVC) 也起到了类似的作用。

PV 可以理解成 Kubernetes 集群中的某个网络存储中对应的一块存储，它与 Volume 很类似，但有以下区别。

- ◎ PV 只能是网络存储，不属于任何 Node，但可以在每个 Node 上访问。
- ◎ PV 并不是定义在 Pod 上的，而是独立于 Pod 之外定义。
- ◎ PV 目前只有几种类型：GCE Persistent Disks、NFS、RBD、iSCSI、AWS ElasticBlockStore、GlusterFS 等。

下面给出了 NFS 类型 PV 的一个 yaml 定义文件，声明了需要 5Gi 的存储空间：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  nfs:
    path: /somepath
    server: 172.17.0.2
```

比较重要的是 PV 的 accessModes 属性，目前有以下类型。

- ◎ **ReadWriteOnce**: 读写权限、并且只能被单个 Node 挂载。
- ◎ **ReadOnlyMany**: 只读权限、允许被多个 Node 挂载。
- ◎ **ReadWriteMany**: 读写权限、允许被多个 Node 挂载。

如果某个 Pod 想申请某种条件的 PV，则首先需要定义一个 **PersistentVolumeClaim** (PVC) 对象：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

然后，在 Pod 的 Volume 定义中引用上述 PVC 即可：

```
volumes:
  - name: mypd
    persistentVolumeClaim:
      claimName: myclaim
```

最后，我们说说 PV 的状态，PV 是有状态的对象，它有以下几种状态。

- ◎ **Available**: 空闲状态。
- ◎ **Bound**: 已经绑定到某个 PVC 上。
- ◎ **Released**: 对应的 PVC 已经删除，但资源还没有被集群收回。
- ◎ **Failed**: PV 自动回收失败。

1.4.11 Namespace (命名空间)

Namespace (命名空间) 是 Kubernetes 系统中的另一个非常重要的概念，Namespace 在很多情况下用于实现多租户的资源隔离。Namespace 通过将集群内部的资源对象“分配”到不同的 Namespace 中，形成逻辑上分组的不同项目、小组或用户组，便于不同的分组在共享使用整个集群的资源的同时还能被分别管理。

Kubernetes 集群在启动后，会创建一个名为“**default**”的 Namespace，通过 kubectl 可以查看到：

```
$ kubectl get namespaces
```

NAME	LABELS	STATUS
default	<none>	Active

接下来，如果不特别指明 Namespace，则用户创建的 Pod、RC、Service 都将被系统创建到这个默认的名为 default 的 Namespace 中。

Namespace 的定义很简单。如下所示的 yaml 定义了名为 development 的 Namespace。

```
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

一旦创建了 Namespace，我们在创建资源对象时就可以指定这个资源对象属于哪个 Namespace。比如在下面的例子中，我们定义了一个名为 busybox 的 Pod，放入 development 这个 Namespace 里：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: development
spec:
  containers:
  - image: busybox
    command:
    - sleep
    - "3600"
    name: busybox
```

此时，使用 kubectl get 命令查看将无法显示：

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
```

这是因为如果不加参数，则 kubectl get 命令将仅显示属于“default”命名空间的资源对象。

可以在 kubectl 命令中加入--namespace 参数来查看某个命名空间中的对象：

```
# kubectl get pods --namespace=development
NAME      READY     STATUS    RESTARTS   AGE
busybox   1/1      Running   0          1m
```

当我们给每个租户创建一个 Namespace 来实现多租户的资源隔离时，还能结合 Kubernetes 的资源配置管理，限定不同租户能占用的资源，例如 CPU 使用量、内存使用量等。关于资源配置管理的问题，在后面的章节中会详细介绍。

1.4.12 Annotation（注解）

Annotation 与 Label 类似，也使用 key/value 键值对的形式进行定义。不同的是 Label 具有严格的命名规则，它定义的是 Kubernetes 对象的元数据（Metadata），并且用于 Label Selector。而 Annotation 则是用户任意定义的“附加”信息，以便于外部工具进行查找，很多时候，Kubernetes 的模块自身会通过 Annotation 的方式标记资源对象的一些特殊信息。

通常来说，用 Annotation 来记录的信息如下。

- ◎ build 信息、release 信息、Docker 镜像信息等，例如时间戳、release id 号、PR 号、镜像 hash 值、docker registry 地址等。
- ◎ 日志库、监控库、分析库等资源库的地址信息。
- ◎ 程序调试工具信息，例如工具名称、版本号等。
- ◎ 团队的联系信息，例如电话号码、负责人名称、网址等。

1.4.13 小结

上述这些组件是 Kubernetes 系统的核心组件，它们共同构成了 Kubernetes 系统的框架和计算模型。通过对它们进行灵活组合，用户就可以快速、方便地对容器集群进行配置、创建和管理。除了本章所介绍的核心组件，在 Kubernetes 系统中还有许多辅助配置的资源对象，例如 LimitRange、ResourceQuota。另外，一些系统内部使用的对象 Binding、Event 等请参考 Kubernetes 的 API 文档。

在第 2 章中，我们将开始深入实践并全面掌握 Kubernetes 的各种使用技巧。

第2章

Kubernetes 实践指南

本章将从 Kubernetes 的系统安装开始，逐步介绍 Kubernetes 的服务相关配置、命令行工具 kubectl 的使用详解，然后通过大量案例实践对 Kubernetes 最核心的容器和微服务架构的概念和用法进行详细说明。

2.1 Kubernetes 安装与配置

2.1.1 安装 Kubernetes

Kubernetes 系统由一组可执行程序组成，用户可以通过 GitHub 上的 Kubernetes 项目页下载编译好的二进制包，或者下载源代码并编译后进行安装。

安装 Kubernetes 对软件和硬件的系统要求如表 2.1 所示。

表 2.1 安装 Kubernetes 对软件和硬件的系统要求

软硬件	最低配置	推荐配置
CPU 和内存	Master: 至少 1 core 和 2GB 内存 Node: 至少 1 core 和 2GB 内存	Master: 2 core 和 4GB 内存 Node: 由于要运行 Docker，所以应根据需要运行的容器数量进行调整
Linux 操作系统	基于 x86_64 架构的各种 Linux 发行版本，包括 Red Hat Linux、CentOS、Fedora、Ubuntu 等，Kernel 版本要求在 3.10 及以上。 也可以在谷歌的 GCE(Google Compute Engine)或者 Amazon 的 AWS (Amazon Web Service) 云平台上进行安装	Red Hat Linux 7 CentOS 7

续表

软硬件	最低配置	推荐配置
Docker	1.9 版本及以上 下载和安装说明见 https://www.docker.com	1.12 版本
etcd	2.0 版本及以上 下载和安装说明见 https://github.com/coreos/etcd/releases	3.0 版本

最简单的安装方法是使用 `yum install kubernetes` 命令完成 Kubernetes 集群的安装，但仍需修改各组件的启动参数，才能完成 Kubernetes 集群的配置。

本章以二进制文件和手工配置启动参数的形式进行安装，对每个组件的配置进行详细说明。

从 Kubernetes 官网下载编译好的二进制包，如图 2.1 所示，下载地址为 <https://github.com/kubernetes/kubernetes/releases>。本书基于 Kubernetes 1.3 版本进行说明。

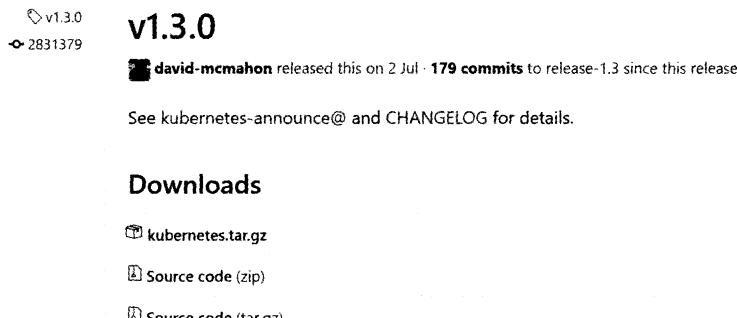


图 2.1 GitHub 上 Kubernetes 的下载页面

在压缩包 `kubernetes.tar.gz` 内包含了 Kubernetes 的服务程序文件、文档和示例。

解压缩后，`server` 子目录中的 `kubernetes-server-linux-amd64.tar.gz` 文件包含了 Kubernetes 需要运行的全部服务程序文件。服务程序文件列表如表 2.2 所示。

表 2.2 服务程序文件列表

文件名	说明
<code>hyperkube</code>	总控程序，用于运行其他 Kubernetes 程序
<code>kube-apiserver</code>	apiserver 主程序
<code>kube-apiserver.docker_tag</code>	apiserver docker 镜像的 tag
<code>kube-apiserver.tar</code>	apiserver docker 镜像文件
<code>kube-controller-manager</code>	controller-manager 主程序
<code>kube-controller-manager.docker_tag</code>	controller-manager docker 镜像的 tag
<code>kube-controller-manager.tar</code>	controller-manager docker 镜像文件
<code>kubectl</code>	客户端命令行工具

续表

文件名	说 明
kubelet	kubelet 主程序
kube-proxy	proxy 主程序
kube-scheduler	scheduler 主程序
kube-scheduler.docker_tag	scheduler docker 镜像的 tag
kube-scheduler.tar	scheduler docker 镜像文件

Kubernetes Master 节点安装部署 etcd、kube-apiserver、kube-controller-manager、kube-scheduler 服务进程。我们使用 kubectl 作为客户端与 Master 进行交互操作，在工作 Node 上仅需部署 kubelet 和 kube-proxy 服务进程。Kubernetes 还提供了一个“all-in-one”的 hyperkube 程序来完成对以上服务程序的启动。

2.1.2 配置和启动 Kubernetes 服务

Kubernetes 的服务都可以通过直接运行二进制文件加上启动参数完成。为了便于管理，常见的做法是将 Kubernetes 服务程序配置为 Linux 的系统开机自动启动的服务。

本节以 CentOS Linux 7 为例，使用 Systemd 系统完成 Kubernetes 服务的配置。其他 Linux 发行版的服务配置请参考相关的系统管理手册。

需要注意的是，CentOS Linux 7 默认启动了 firewalld——防火墙服务，而 Kubernetes 的 Master 与工作 Node 之间会有大量的网络通信，安全的做法是在防火墙上配置各组件需要相互通信的端口号，具体要配置的端口号详见 2.1.6 节中各服务监听的端口号说明。在一个安全的内部网络环境中可以关闭防火墙服务：

```
# systemctl disable firewalld
# systemctl stop firewalld
```

将 Kubernetes 的可执行文件复制到 /usr/bin（如果复制到其他目录，则将 systemd 服务文件中的文件路径修改正确即可），然后对服务进行配置。

在下面的服务启动参数说明中主要介绍最重要的启动参数，每个服务的启动参数还有很多，详见 2.1.6 节的完整说明。有兴趣的读者可以尝试修改它们，以观察服务运行的不同效果。

1. Master 上的 etcd、kube-apiserver、kube-controller-manager、kube-scheduler 服务

1) etcd 服务

etcd 服务作为 Kubernetes 集群的主数据库，在安装 Kubernetes 各服务之前需要首先安装和启动。

从 GitHub 官网下载 etcd 发布的二进制文件，将 etcd 和 etcdctl 文件复制到 /usr/bin 目录。

设置 systemd 服务文件 /usr/lib/systemd/system/etcd.service：

```
[Unit]
Description=Etcd Server
After=network.target

[Service]
Type=simple
WorkingDirectory=/var/lib/etcd/
EnvironmentFile=-/etc/etcd/etcd.conf
ExecStart=/usr/bin/etcd

[Install]
WantedBy=multi-user.target
```

其中 WorkingDirectory (/var/lib/etcd/) 表示 etcd 数据保存的目录，需要在启动 etcd 服务之前进行创建。

配置文件 /etc/etcd/etcd.conf 通常不需要特别的参数设置（详细的参数配置内容参见官方文档），etcd 默认将监听在 http://127.0.0.1:2379 地址供客户端连接。

配置完成后，通过 systemctl start 命令启动 etcd 服务。同时，使用 systemctl enable 命令将服务加入开机启动列表中：

```
# systemctl daemon-reload
# systemctl enable etcd.service
# systemctl start etcd.service
```

通过执行 etcdctl cluster-health，可以验证 etcd 是否正确启动：

```
# etcdctl cluster-health
member ce2a822cea30bfca is healthy: got healthy result from http://127.0.0.1:2379
cluster is healthy
```

2) kube-apiserver 服务

将 kube-apiserver 的可执行文件复制到 /usr/bin 目录。

编辑 systemd 服务文件 /usr/lib/systemd/system/kube-apiserver.service，内容如下：

```
[Unit]
Description=Kubernetes API Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=etcd.service
Wants=etcd.service

[Service]
EnvironmentFile=/etc/kubernetes/apiserver
```

```
ExecStart=/usr/bin/kube-apiserver $KUBE_API_ARGS
Restart=on-failure
Type=notify
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件/etc/kubernetes/apiserver 的内容包括了 kube-apiserver 的全部启动参数，主要的配置参数在变量 KUBE_API_ARGS 中指定。

```
# cat /etc/kubernetes/apiserver
KUBE_API_ARGS="--etcd_servers=http://127.0.0.1:2379
--insecure-bind-address=0.0.0.0 --insecure-port=8080
--service-cluster-ip-range=169.169.0.0/16 --service-node-port-range=1-65535
--admission_control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota --logtostderr=false --log-dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- ◎ --etcd_servers: 指定 etcd 服务的 URL。
- ◎ --insecure-bind-address: apiserver 绑定主机的非安全 IP 地址，设置 0.0.0.0 表示绑定所有 IP 地址。
- ◎ --insecure-port: apiserver 绑定主机的非安全端口号，默认为 8080。
- ◎ --service-cluster-ip-range: Kubernetes 集群中 Service 的虚拟 IP 地址段范围，以 CIDR 格式表示，例如 169.169.0.0/16，该 IP 范围不能与物理机的真实 IP 段有重合。
- ◎ --service-node-port-range: Kubernetes 集群中 Service 可映射的物理机端口号范围，默认为 30000~32767。
- ◎ --admission_control: Kubernetes 集群的准入控制设置，各控制模块以插件的形式依次生效。
- ◎ --logtostderr: 设置为 false 表示将日志写入文件，不写入 stderr。
- ◎ --log-dir: 日志目录。
- ◎ --v: 日志级别。

3) kube-controller-manager 服务

kube-controller-manager 服务依赖于 kube-apiserver 服务。

```
# cat /usr/lib/systemd/system/kube-controller-manager.service
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=kube-apiserver.service
Requires=kube-apiserver.service
```

```
[Service]
EnvironmentFile=/etc/kubernetes/controller-manager
ExecStart=/usr/bin/kube-controller-manager $KUBE_CONTROLLER_MANAGER_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件/etc/kubernetes/controller-manager 的内容包括了 kube-controller-manager 的全部启动参数，主要的配置参数在变量 KUBE_CONTROLLER_MANAGER_ARGS 中指定。

```
# cat /etc/kubernetes/controller-manager
KUBE_CONTROLLER_MANAGER_ARGS="--master=http://192.168.18.3:8080
--logtostderr=false --log-dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- ◎ **--master:** 指定 apiserver 的 URL 地址。
- ◎ **--logtostderr:** 设置为 false 表示将日志写入文件，不写入 stderr。
- ◎ **--log-dir:** 日志目录。
- ◎ **--v:** 日志级别。

4) kube-scheduler 服务

kube-scheduler 服务也依赖于 kube-apiserver 服务。

```
# cat /usr/lib/systemd/system/kube-controller-manager.service
[Unit]
Description=Kubernetes Controller Manager
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=kube-apiserver.service
Requires=kube-apiserver.service

[Service]
EnvironmentFile=/etc/kubernetes/scheduler
ExecStart=/usr/bin/kube-scheduler $KUBE_SCHEDULER_ARGS
Restart=on-failure
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

配置文件/etc/kubernetes/scheduler 的内容包括了 kube-scheduler 的全部启动参数，主要的配置参数在变量 KUBE_SCHEDULER_ARGS 中指定。

```
# cat /etc/kubernetes/scheduler
```

```
KUBE_SCHEDULER_ARGS="--master=http://192.168.18.3:8080 --logtostderr=false
--log-dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- ◎ **--master:** 指定 apiserver 的 URL 地址。
- ◎ **--logtostderr:** 设置为 false 表示将日志写入文件，不写入 stderr。
- ◎ **--log-dir:** 日志目录。
- ◎ **--v:** 日志级别。

配置完成后，执行 `systemctl start` 命令按顺序启动这 3 个服务。同时，使用 `systemctl enable` 命令将服务加入开机启动列表中。

```
# systemctl daemon-reload
# systemctl enable kube-apiserver.service
# systemctl start kube-apiserver.service
# systemctl enable kube-controller-manager
# systemctl start kube-controller-manager
# systemctl enable kube-scheduler
# systemctl start kube-scheduler
```

通过 `systemctl status <service_name>` 来验证服务的启动状态，“running”表示启动成功。

至此，Master 上所需的服务就全部启动完成了。

2. Node 上的 kubelet、kube-proxy 服务

在工作 Node 节点上需要预先安装好 Docker Daemon 并且正常启动。Docker 的安装详见 <http://www.docker.com> 的说明。

1) kubelet 服务

kubelet 服务依赖于 Docker 服务。

```
# cat /usr/lib/systemd/system/kubelet.service
[Unit]
Description=Kubernetes Kubelet Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=docker.service
Requires=docker.service

[Service]
WorkingDirectory=/var/lib/kubelet
EnvironmentFile=/etc/kubernetes/kubelet
ExecStart=/usr/bin/kubelet $KUBELET_ARGS
Restart=on-failure
```

```
[Install]
WantedBy=multi-user.target
```

其中 WorkingDirectory 表示 kubelet 保存数据的目录，需要在启动 kubelet 服务之前进行创建。

配置文件/etc/kubernetes/kubelet 的内容包括了 kubelet 的全部启动参数，主要的配置参数在变量 KUBELET_ARGS 中指定。

```
# cat /etc/kubernetes/kubelet
KUBELET_ARGS="--api-servers=http://192.168.18.3:8080
--hostname-override=192.168.18.3 --logtostderr=false
--log-dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- ◎ **--api-servers:** 指定 apiserver 的 URL 地址，可以指定多个。
- ◎ **--hostname-override:** 设置本 Node 的名称。
- ◎ **--logtostderr:** 设置为 false 表示将日志写入文件，不写入 stderr。
- ◎ **--log-dir:** 日志目录。
- ◎ **--v:** 日志级别。

2) kube-proxy 服务

kube-proxy 服务依赖于 network 服务。

```
[Unit]
Description=Kubernetes Kube-Proxy Server
Documentation=https://github.com/GoogleCloudPlatform/kubernetes
After=network.target
Requires=network.service
```

```
[Service]
EnvironmentFile=/etc/kubernetes/proxy
ExecStart=/usr/bin/kube-proxy $KUBE_PROXY_ARGS
Restart=on-failure
LimitNOFILE=65536
```

```
[Install]
WantedBy=multi-user.target
```

配置文件/etc/kubernetes/proxy 的内容包括了 kube-proxy 的全部启动参数，主要的配置参数在变量 KUBE_PROXY_ARGS 中指定。

```
# cat /etc/kubernetes/proxy
KUBE_PROXY_ARGS="--master=http://192.168.18.3:8080 --logtostderr=false
--log-dir=/var/log/kubernetes --v=2"
```

对启动参数的说明如下。

- ◎ **--master:** 指定 apiserver 的 URL 地址。
- ◎ **--logtostderr:** 设置为 false 表示将日志写入文件，不写入 stderr。
- ◎ **--log-dir:** 日志目录。
- ◎ **--v:** 日志级别。

配置完成后，通过 systemctl 启动 kubelet 和 kube-proxy 服务：

```
# systemctl daemon-reload
# systemctl enable kubelet.service
# systemctl start kubelet.service
# systemctl enable kube-proxy
# systemctl start kube-proxy
```

kubelet 默认采用向 Master 自动注册本 Node 的机制，在 Master 上查看各 Node 的状态，状态为 Ready 表示 Node 已经成功注册并且状态为可用。

```
# kubectl get nodes
NAME           STATUS    AGE
192.168.18.3  Ready     1m
```

等所有 Node 的状态都为 Ready 之后，一个 Kubernetes 集群就启动完成了。接下来就可以创建 Pod、RC、Service 等资源对象来部署 Docker 容器应用了。

2.1.3 Kubernetes 集群的安全设置

1. 基于 CA 签名的双向数字证书认证方式

在一个安全的内网环境中，Kubernetes 的各个组件与 Master 之间可以通过 apiserver 的非安全端口 `http://apiserver:8080` 进行访问。但如果 apiserver 需要对外提供服务，或者集群中的某些容器也需要访问 apiserver 以获取集群中的某些信息，则更安全的做法是启用 HTTPS 安全机制。Kubernetes 提供了基于 CA 签名的双向数字证书认证方式和简单的基于 HTTP BASE 或 TOKEN 的认证方式，其中 CA 证书方式的安全性最高。本节先介绍以 CA 证书的方式配置 Kubernetes 集群，要求 Master 上的 `kube-apiserver`、`kube-controller-manager`、`kube-scheduler` 进程及各 Node 上的 `kubelet`、`kube-proxy` 进程进行 CA 签名双向数字证书安全设置。

基于 CA 签名的双向数字证书的生成过程如下。

- (1) 为 `kube-apiserver` 生成一个数字证书，并用 CA 证书进行签名。
- (2) 为 `kube-apiserver` 进程配置证书相关的启动参数，包括 CA 证书（用于验证客户端证书）

的签名真伪)、自己的经过 CA 签名后的证书及私钥。

(3) 为每个访问 Kubernetes API Server 的客户端(如 kube-controller-manager、kube-scheduler、kubelet、kube-proxy 及调用 API Server 的客户端程序 kubectl 等) 进程生成自己的数字证书，也都用 CA 证书进行签名，在相关程序的启动参数里增加 CA 证书、自己的证书等相关参数。

1) 设置 kube-apiserver 的 CA 证书相关的文件和启动参数

使用 OpenSSL 工具在 Master 服务器上创建 CA 证书和私钥相关的文件：

```
# openssl genrsa -out ca.key 2048
# openssl req -x509 -new -nodes -key ca.key -subj "/CN=yourcompany.com" -days
5000 -out ca.crt
# openssl genrsa -out server.key 2048
```

注意：生成 ca.crt 时，-subj 参数中 “/CN” 的值通常为域名。

准备 master_ssl.cnf 文件，该文件用于 x509 v3 版本的证书。在该文件中主要需要设置 Master 服务器的 hostname (k8s-master)、IP 地址 (192.168.18.3)，以及 Kubernetes Master Service 的虚拟服务名称 (kubernetes.default 等) 和该虚拟服务的 ClusterIP 地址 (169.169.0.1)。

master_ssl.cnf 文件的示例如下：

```
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name
[req_distinguished_name]
[v3_req]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names
[alt_names]
DNS.1 = kubernetes
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
DNS.4 = kubernetes.default.svc.cluster.local
DNS.5 = k8s-master
IP.1 = 169.169.0.1
IP.2 = 192.168.18.3
```

基于 master_ssl.cnf 创建 server.csr 和 server.crt 文件。在生成 server.csr 时，-subj 参数中“/CN”指定的名字需为 Master 所在的主机名。

```
# openssl req -new -key server.key -subj "/CN=k8s-master" -config master_ssl.cnf
-out server.csr
# openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -days
5000 -extensions v3_req -extfile master_ssl.cnf -out server.crt
```

全部执行完后会生成 6 个文件：ca.crt、ca.key、ca.srl、server.crt、server.csr、server.key。

将这些文件复制到一个目录中（例如/var/run/kubernetes/），然后设置 kube-apiserver 的三个启动参数“--client-ca-file”“--tls-cert-file”和“--tls-private-key-file”，分别代表 CA 根证书文件、服务端证书文件和服务端私钥文件：

```
--client_ca_file=/var/run/kubernetes/ca.crt
--tls_private_key_file=/var/run/kubernetes/server.key
--tls_cert_file=/var/run/kubernetes/server.crt
```

同时，可以关掉非安全端口 8080，设置安全端口为 443（默认为 6443）：

```
--insecure_port=0
--secure_port=443
```

最后重启 kube-apiserver 服务。

2) 设置 kube-controller-manager 的客户端证书、私钥和启动参数

```
$ openssl genrsa -out cs_client.key 2048
$ openssl req -new -key cs_client.key -subj "/CN=k8s-node-1" -out cs_client.csr
$ openssl x509 -req -in cs_client.csr -CA ca.crt -CAkey ca.key -CAcreateserial
-out cs_client.crt -days 5000
```

其中，在生成 cs_client.crt 时，-CA 参数和-CAkey 参数使用的是 apiserver 的 ca.crt 和 ca.key 文件。然后将这些文件复制到一个目录中（例如/var/run/kubernetes/）。

接下来创建/etc/kubernetes/kubeconfig 文件（kube-controller-manager 与 kube-scheduler 共用），配置客户端证书等相关参数，内容如下：

```
apiVersion: v1
kind: Config
users:
- name: controllermanager
  user:
    client_certificate: /var/run/kubernetes/cs_client.crt
    client_key: /var/run/kubernetes/cs_client.key
clusters:
- name: local
  cluster:
    certificate_authority: /var/run/kubernetes/ca.crt
contexts:
- context:
    cluster: local
    user: controllermanager
    name: my-context
current-context: my-context
```

然后，设置 kube-controller-manager 服务的启动参数，注意，--master 的地址为 HTTPS 安全服务地址，不使用非安全地址 http://192.168.18.3:8080。

```
--master=https://192.168.18.3:443  
--service_account_private_key_file=/var/run/kubernetes/server.key  
--root-ca-file=/var/run/kubernetes/ca.crt  
--kubeconfig=/etc/kubernetes/kubeconfig
```

重启 kube-controller-manager 服务。

3) 设置 kube-scheduler 启动参数

kube-scheduler 复用上一步 kube-controller-manager 创建的客户端证书，配置启动参数：

```
--master=https://192.168.18.3:443  
--kubeconfig=/etc/kubernetes/kubeconfig
```

重启 kube-scheduler 服务。

4) 设置每台 Node 上 kubelet 的客户端证书、私钥和启动参数

首先复制 kube-apiserver 的 ca.crt 和 ca.key 文件到 Node 上，在生成 kubelet_client.crt 时-CA 参数和-CAkey 参数使用的是 apiserver 的 ca.crt 和 ca.key 文件。在生成 kubelet_client.csr 时-subj 参数中的 “/CN” 设置为本 Node 的 IP 地址。

```
$ openssl genrsa -out kubelet_client.key 2048  
$ openssl req -new -key kubelet_client.key -subj "/CN=192.168.18.4" -out kubelet_client.csr  
$ openssl x509 -req -in kubelet_client.csr -CA ca.crt -CAkey ca.key  
-CAcreateserial -out kubelet_client.crt -days 5000
```

将这些文件复制到一个目录中（例如/var/run/kubernetes/）。

接下来创建/etc/kubernetes/kubeconfig 文件（kubelet 和 kube-proxy 进程共用），配置客户端证书等相关参数，内容如下：

```
apiVersion: v1  
kind: Config  
users:  
- name: kubelet  
  user:  
    client-certificate: /etc/kubernetes/ssl_keys/kubelet_client.crt  
    client-key: /etc/kubernetes/ssl_keys/kubelet_client.key  
clusters:  
- name: local  
  cluster:  
    certificate-authority: /etc/kubernetes/ssl_keys/ca.crt  
contexts:  
- context:  
  cluster: local  
  user: kubelet  
  name: my-context  
current-context: my-context
```

然后，设置 kubelet 服务的启动参数：

```
--api_servers=https://192.168.18.3:443
--kubeconfig=/etc/kubelet/kubeconfig
```

最后重启 kubelet 服务。

5) 设置 kube-proxy 的启动参数

kube-proxy 复用上一步 kubelet 创建的客户端证书，配置启动参数：

```
--master=https://192.168.18.3:443
--kubeconfig=/etc/kubernetes/kubeconfig
```

重启 kube-proxy 服务。

至此，一个基于 CA 的双向数字证书认证的 Kubernetes 集群环境就搭建完成了。

6) 设置 kubectl 客户端使用安全方式访问 apiserver

在使用 kubectl 对 Kubernetes 集群进行操作时，默认使用非安全端口 8080 对 apiserver 进行访问，也可以设置为安全访问 apiserver 的模式，需要设置 3 个证书相关的参数“—certificate-authority”“--client-certificate”和“--client-key”，分别表示用于 CA 授权的证书、客户端证书和客户端密钥。

- ◎ --certificate-authority：使用为 kube-apiserver 生成的 ca.crt 文件。
- ◎ --client-certificate：使用为 kube-controller-manager 生成的 cs_client.crt 文件。
- ◎ --client-key：使用为 kube-controller-manager 生成的 cs_client.key 文件。

同时，指定 apiserver 的 URL 地址为 HTTPS 安全地址（例如 https://k8s-master:443），最后输入需要执行的子命令，即可对 apiserver 进行安全访问了：

```
# kubectl --server=https://k8s-master:443
--certificate-authority=/etc/kubernetes/ssl_keys/ca.crt
--client-certificate=/etc/kubernetes/ssl_keys/cs_client.crt
--client-key=/etc/kubernetes/ssl_keys/cs_client.key get nodes
NAME          STATUS     AGE
k8s-node-1    Ready      1h
```

2. 基于 HTTP BASE 或 TOKEN 的简单认证方式

除了基于 CA 的双向数字证书认证方式，Kubernetes 也提供了基于 HTTP BASE 或 TOKEN 的简单认证方式。各组件与 apiserver 之间的通信方式仍然采用 HTTPS，但不使用 CA 数字证书。

采用基于 HTTP BASE 或 TOKEN 的简单认证方式时，API Server 对外暴露 HTTPS 端口，客户端提供用户名、密码或 Token 来完成认证过程。需要说明的是，kubectl 命令行工具比较特殊，它同时支持 CA 双向认证与简单认证两种模式与 apiserver 通信，其他客户端组件只能配置

为双向安全认证或非安全模式与 apiserver 通信。

基于 HTTP BASE 认证的配置过程如下。

(1) 创建包括用户名、密码和 UID 的文件 basic_auth_file，放置在合适的目录中，例如 /etc/kubernetes 目录。需要注意的是，这是一个纯文本文件，用户名、密码都是明文。

```
# vi /etc/kubernetes/basic_auth_file
admin,admin,1
system,system,2
```

(2) 设置 kube-apiserver 的启动参数 “--basic_auth_file”，使用上述文件提供安全认证：

```
--secure-port=443
--basic_auth_file=/etc/kubernetes/basic_auth_file
```

然后，重启 API Server 服务。

(3) 使用 kubectl 通过指定的用户名和密码来访问 API Server：

```
# kubectl --server=https://192.168.18.3:443 --username=admin --password=admin
--insecure-skip-tls-verify=true get nodes
```

基于 TOKEN 认证的配置过程如下。

(1) 创建包括用户名、密码和 UID 的文件 token_auth_file，放置在合适的目录中，例如 /etc/kubernetes 目录。需要注意的是，这是一个纯文本文件，用户名、密码都是明文。

```
$ cat /etc/kubernetes/token_auth_file
admin,admin,1
system,system,2
```

(2) 设置 kube-apiserver 的启动参数 “--token_auth_file”，使用上述文件提供安全认证：

```
--secure-port=443
--token_auth_file=/etc/kubernetes/token_auth_file
```

然后，重启 API Server 服务。

(3) 用 curl 验证和访问 API Server：

```
$ curl -k --header "Authorization:Bearer admin" https://192.168.18.3:443/version
{
  "major": "1",
  "minor": "3",
  "gitVersion": "v1.3.3",
  "gitCommit": "c6411395e09da356c608896d3d9725acab821418",
  "gitTreeState": "clean",
  "buildDate": "2016-07-22T20:22:25Z",
  "goVersion": "go1.6.2",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

2.1.4 Kubernetes的版本升级

Kubernetes的版本升级需要考虑到当前集群中正在运行的容器不受影响。应对集群中的各Node逐个进行隔离，然后等待在其上运行的容器全部执行完成，再更新该Node上的kubelet和kube-proxy服务，将全部Node都更新完成后，最后更新Master的服务。

- ◎ 通过官网获取最新版本的二进制包kubernetes.tar.gz，解压缩后提取服务二进制文件。
- ◎ 逐个隔离Node，等待在其上运行的全部容器工作完成，更新kubelet和kube-proxy服务文件，然后重启这两个服务。
- ◎ 更新Master的kube-apiserver、kube-controller-manager、kube-scheduler服务文件并重启。

2.1.5 内网中的Kubernetes相关配置

Kubernetes在能够访问Internet网络的环境中使用起来非常方便，一方面在docker.io和gcr.io网站中已经存在了大量官方制作的Docker镜像，另一方面GCE、AWS提供的云平台已经很成熟了，用户通过租用一定的空间来部署Kubernetes集群也很简便。

但是，许多企业内部由于安全性的原因无法访问Internet。对于这些企业就需要通过创建一个内部的私有Docker Registry，并修改一些Kubernetes的配置，来启动内网中的Kubernetes集群。

1. Docker Private Registry（私有Docker镜像库）

使用Docker提供的Registry镜像创建一个私有镜像仓库。

详细的安装步骤请参考Docker的官方文档<https://docs.docker.com/registry/deploying/>。

2. kubelet配置

由于在Kubernetes中是以Pod而不是Docker容器为管理单元的，在kubelet创建Pod时，还通过启动一个名为google_containers/pause的镜像来实现Pod的概念。

该镜像存在于谷歌镜像库<http://gcr.io>中，需要通过一台能够连上Internet的服务器将其下载，导出文件，再push到私有Docker Registry中去。

之后，可以给每台Node的kubelet服务的启动参数加上--pod_infra_container_image参数，指定为私有Docker Registry中pause镜像的地址。例如：

```
# cat /etc/kubernetes/kubelet
KUBELET_ARGS="--api-servers=http://192.168.18.3:8080
--hostname-override=192.168.18.3 --log-dir=/var/log/kubernetes --v=2
```

```
--pod_infra_container_image=gcr.io/google_containers/pause-amd64:3.0"
```

如果该镜像无法下载，则也可以从 Docker Hub 上进行下载：

```
# docker pull kubeguide/google_containers/pause-amd64:3.0
```

修改 kubelet 配置文件中的 `--pod_infra_container_image` 参数如下：

```
--pod_infra_container_image=kubeguide/google_containers/pause-amd64:3.0
```

然后重启 kubelet 服务：

```
# systemctl restart kubelet
```

通过以上设置就在内网环境中搭建了一个企业内部的私有容器云平台。

2.1.6 Kubernetes 核心服务配置详解

我们在 2.1.2 节对 Kubernetes 各服务启动进程的关键配置参数进行了简要说明，实际上 Kubernetes 的每个服务都提供了许多可配置的参数。这些参数涉及安全性、性能优化及功能扩展（Plugin）等方方面面。全面理解和掌握这些参数的含义和配置，无论对于 Kubernetes 的生产部署还是日常运维都有很好的帮助。

每个服务的可用参数都可以通过运行“`cmd --help`”命令进行查看，其中 `cmd` 为具体的服务启动命令，例如 `kube-apiserver`、`kube-controller-manager`、`kube-scheduler`、`kubelet`、`kube-proxy` 等。另外，也可以通过在命令的配置文件（例如`/etc/kubernetes/kubelet` 等）中添加“`--参数名=参数取值`”的语句来完成对某个参数的配置。

本节将对 Kubernetes 所有服务的参数进行全面介绍，为了方便学习和查阅，对每个服务的参数用一个小节进行详细说明。

1. 公共配置参数

公共配置参数适用于所有服务，如表 2.3 所示的参数可用于 `kube-apiserver`、`kube-controller-manager`、`kube-scheduler`、`kubelet`、`kube-proxy`。本节对这些参数进行统一说明，不再在每个服务的参数列表中列出。

表 2.3 公共配置参数表

参数名和取值示例	说 明
<code>--log-backtrace-at=:0</code>	记录日志每到“file:行号”时打印一次 stack trace
<code>--log-dir=</code>	日志文件路径
<code>--log-flush-frequency=5s</code>	设置 flush 日志文件的时间间隔
<code>--logtostderr=true</code>	设置为 true 则表示将日志输出到 stderr，不输出到日志文件

续表

参数名和取值示例	说 明
--alsologtostderr=false	设置为 true 则表示将日志输出到文件的同时输出到 stderr
--stderrthreshold=2	将该 threshold 级别之上的日志输出到 stderr
--v=0	glog 日志级别
--vmodule=	glog 基于模块的详细日志级别
--version=[false]	设置为 true 则将打印版本信息然后退出

2. kube-apiserver 启动参数

对 kube-apiserver 启动参数的详细说明如表 2.4 所示。

表 2.4 对 kube-apiserver 启动参数的详细说明

参数名和取值示例	说 明
--admission-control="AlwaysAdmit"	<p>对发送给 API Server 的任何请求进行准入控制，配置为一个“准入控制器”的列表，多个准入控制器时以逗号分隔。多个准入控制器将按顺序对发送给 API Server 的请求进行拦截和过滤，若某个准入控制器不允许该请求通过，则 API Server 拒绝此调用请求。可配置的准入控制器如下。</p> <ul style="list-style-type: none"> ① AlwaysAdmit：允许所有请求。 ② AlwaysPullImages：在启动容器之前总是去下载镜像，相当于在每个容器的配置项 imagePullPolicy=Always。 ③ AlwaysDeny：禁止所有请求，一般用于测试。 ④ DenyExecOnPrivileged：它会拦截所有想在 privileged container 上执行命令的请求。如果你的集群支持 privileged container，你又希望限制用户在这些 privileged container 上执行命令，那么强烈推荐你使用它。 ⑤ ServiceAccount：这个 plug-in 将 serviceAccounts 实现了自动化，如果你想要使用 ServiceAccount 对象，那么强烈推荐你使用它。 ⑥ SecurityContextDeny：这个插件将使用了 SecurityContext 的 Pod 中定义的选项全部失效。SecurityContext 在 container 中定义了操作系统级别的安全设定 (uid、gid、capabilities、SELinux 等)。 ⑦ ResourceQuota：用于配额管理目的，作用于 Namespace 上，它会观察所有的请求，确保在 namespace 上的配额不会超标。推荐在 admission control 参数列表中这个插件排最后一个。 ⑧ LimitRanger：用于配额管理，作用于 Pod 与 Container 上，确保 Pod 与 Container 上的配额不会超标。 ⑨ NamespaceExists（已过时）：对所有请求校验 namespace 是否已存在，如果不存在则拒绝请求。已合并至 NamespaceLifecycle。 ⑩ NamespaceAutoProvision（已过时）：对所有请求校验 namespace，如果不存在则自动创建该 namespace，推荐使用 NamespaceLifecycle。

续表

参数名和取值示例	说 明
--admission-control="AlwaysAdmit"	<p>NamespaceLifecycle: 如果尝试在一个不存在的 namespace 中创建资源对象，则该创建请求将被拒绝。当删除一个 namespace 时，系统将会删除该 namespace 中的所有对象，包括 Pod、Service 等。</p> <p>如果启用多种准入选项，则建议加载的顺序是：</p> <pre>--admission-control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota</pre>
--admission-control-config-file=""	与准入控制规则相关的配置文件
--advertise-address=<nil>	用于广播给集群的所有成员自己的 IP 地址，不指定该地址将使用“--bind-address”定义的 IP 地址
--allow-privileged=false	如果设置为 true，则 Kubernetes 将允许在 Pod 中运行拥有系统特权的容器应用，与 docker run --privileged 的功效相同
--apiserver-count=1	集群中运行的 API Server 数量
--authentication-token-webhook-cache-ttl=2m0s	将 webhook token authenticator 返回的响应保存在缓存内的时间，默认为两分钟
--authentication-token-webhook-config-file=""	Webhook 相关的配置文件，将用于 token authentication
--authorization-mode="AlwaysAllow"	到 API Server 的安全访问的认证模式列表，以逗号分隔，可选值包括：AlwaysAllow、AlwaysDeny、ABAC、Webhook、RBAC
--authorization-policy-file=""	当--authorization-mode 设置为 ABAC 时使用的 csv 格式的授权配置文件
--authorization-rbac-super-user=""	当--authorization-mode 设置为 RBAC 时使用的超级用户名，使用该用户名可以不进行 RBAC 认证
--authorization-webhook-cache-authorized-ttl=5m0s	将 webhook authorizer 返回的“已授权”响应保存在缓存内的时间，默认为 5 分钟。
--authorization-webhook-cache-unauthorized-ttl=30s	将 webhook authorizer 返回的“未授权”响应保存在缓存内的时间，默认为 30 秒
--authorization-webhook-config-file=""	当--authorization-mode 设置为 webhook 时使用的授权配置文件
--basic-auth-file=""	设置该文件用于通过 HTTP 基本认证的方式访问 API Server 的安全端口
--bind-address=0.0.0.0	Kubernetes API Server 在本地地址的 6443 端口开启安全的 HTTPS 服务，默认为 0.0.0.0
--cert-dir="/var/run/kubernetes"	TLS 证书所在的目录，默认为 /var/run/kubernetes。如果设置了--tls-cert-file 和--tls-private-key-file，则该设置将被忽略
--client-ca-file=""	如果指定，则该客户端证书将被用于认证过程
--cloud-config=""	云服务商的配置文件路径，不配置则表示不使用云服务商的配置文件
--cloud-provider=""	云服务商的名称，不配置则表示不使用云服务商
--cors-allowed-origins=[]	CORS（跨域资源共享）设置允许访问的源域列表，用逗号分隔，并可使用正则表达式匹配子网。如果不指定，则表示不启用 CORS
--delete-collection-workers=1	启动 DeleteCollection 的工作线程数，用于提高清理 namespace 的效率
--deserialization-cache-size=50000	设置内存中缓存的 JSON 对象的个数

续表

参数名和取值示例	说 明
--enable-garbage-collector[=false]	设置为 true 表示启用垃圾回收器。必须与 kube-controller-manager 的该参数设置为相同的值
--enable-swagger-ui[=false]	设置为 true 表示启用 swagger ui 网页，可通过 API Server 的 URL/swagger-ui 访问
--etcd-cafile=""	到 etcd 安全连接使用的 SSL CA 文件
--etcd-certfile=""	到 etcd 安全连接使用的 SSL 证书文件
--etcd-keyfile=""	到 etcd 安全连接使用的 SSL key 文件
--etcd-prefix="/registry"	在 etcd 中保存 Kubernetes 集群数据的根目录名，默认为/registry
--etcd-quorum-read[=false]	设置为 true 表示启用 quorum read 机制
--etcd-servers=[]	以逗号分隔的 etcd 服务 URL 列表，etcd 服务以 http://ip:port 格式表示
--etcd-servers-overrides=[]	按资源覆盖 etcd 服务的设置，以逗号分隔。单个覆盖格式为：group/resource #servers，其中 servers 格式为 http://ip:port，以分号分隔
--event-ttl=1h0m0s	Kubernetes API Server 中各种事件（通常用于审计和追踪）在系统中保存的时间，默认为 1 小时
--experimental-keystone-url=""	设置 keystone 鉴权插件地址，实验用
--external-hostname=""	用于生成该 Master 的对外 URL 地址，例如用于 swagger api 文档中的 URL 地址。
--insecure-bind-address=127.0.0.1	绑定的不安全 IP 地址，与--insecure-port 共同使用，默认为 localhost。设置为 0.0.0.0 表示使用全部网络接口
--insecure-port=8080	提供非安全认证访问的监听端口，默认为 8080。应在防火墙中进行配置，以使得外部客户端不可以通过非安全端口访问 API Server
--ir-data-source="influxdb"	设置 InitialResources 使用的数据源，可配置项包括 influxdb、gcm
--ir-dbname="k8s"	InitialResources 所需指标保存在 influxdb 中的数据库名称，默认为 k8s
--ir-hawkular=""	设置 Hawkular 的 URL 地址
--ir-influxdb-host="localhost:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb:api"	InitialResources 所需指标所在 influxdb 的 URL 地址，默认为 localhost:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-influxdb:api
--ir-namespace-only[=false]	设置为 true 表示从相同的 namespace 内的数据进行估算
--ir-password="root"	连接 influxdb 数据库的密码
--ir-percentile=90	InitialResources 进行资源估算时的采样百分比，实验用
--ir-user="root"	连接 influxdb 数据库的用户名
--kubelet-certificate-authority=""	用于 CA 授权的 cert 文件路径
--kubelet-client-certificate=""	用于 TLS 的客户端证书文件路径
--kubelet-client-key=""	用于 TLS 的客户端 key 文件路径
--kubelet-https[=true]	指定 kubelet 是否使用 HTTPS 连接
--kubelet-timeout=5s	kubelet 执行操作的超时时间
--kubernetes-service-node-port=0	设置 Master 服务是否使用 NodePort 模式，如果设置，则 Master 服务将映射到物理机的端口号；设置为 0 表示以 ClusterIP 的形式启动 Master 服务

续表

参数名和取值示例	说 明
--long-running-request-regexp="(^)(watch proxy)(/\$) (logs? portforward exec attach)?\$)"	以正则表达式配置哪些需要长时间执行的请求不会被系统进行超时处理
--master-service-namespace="default"	设置 Master 服务所在的 namespace， 默认为 default
--max-connection-bytes-per-sec=0	设置为非 0 的值表示限制每个客户端连接的带宽为 xx 字节/秒， 目前仅用于需要长时间执行的请求
--max-requests-inflight=400	同时处理的最大请求数量， 默认为 400， 超过该数量的请求将被拒绝。 设置为 0 表示无限制
--min-request-timeout=1800	最小请求处理超时时间， 单位为秒， 默认为 1800 秒， 目前仅用于 watch request handler， 其将会在该时间值上加一个随机时间作为请求的超时时间
--oidc-ca-file=""	该文件内设置鉴权机构， OpenID Server 的证书将被其中一个机构进行验证。如果不设置，则将使用主机的 root CA 证书
--oidc-client-id=""	OpenID Connect 的客户端 ID，在 oidc-issuer-url 设置时必须设置
--oidc-groups-claim=""	定制的 OpenID Connect 用户组声明的设置，以字符串数组的形式表示，实验用
--oidc-issuer-url=""	OpenID 发行者的 URL 地址，仅支持 HTTPS scheme，用于验证 OIDC JSON Web Token (JWT)
--oidc-username-claim="sub"	OpenID claim 的用户名， 默认为 “sub”，实验用
--profiling=true	打开性能分析，可以通过 <host>:<port>/debug/pprof 地址查看程序栈、线程等系统信息
--repair-malformed-updates[=true]	设置为 true 表示服务器将尽可能修复无效或格式错误的 update request，以通过正确性校验，例如在一个 update request 中将一个已存在的 UID 值设置为空
--runtime-config=	一组 key=value 用于运行时的配置信息。apis/<groupVersion>/<resource> 可用于打开或关闭对某个 API 版本的支持。api/all 和 api/legacy 特别用于支持所有版本的 API 或支持旧版本的 API
--secure-port=6443	设置 API Server 使用的 HTTPS 安全模式端口号，设置为 0 表示不启用 HTTPS
--service-account-key-file=""	包含 PEM-encoded x509 RSA 公钥和私钥的文件路径，用于验证 Service Account 的 token。不指定则使用--tls-private-key-file 指定的文件
--service-account-lookup[=false]	设置为 true 时，系统会到 etcd 验证 ServiceAccount token 是否存在
--service-cluster-ip-range=<nil>	Service 的 Cluster IP (虚拟 IP) 池，例如 169.169.0.0/16，这个 IP 地址池不能与物理机所在的网络重合
--service-node-port-range=	Service 的 NodePort 能使用的主机端口号范围， 默认为 30000~32767，包括 30000 和 32767
--ssh-keyfile=""	如果指定，则通过 SSH 使用指定的秘钥文件对 Node 进行访问
--ssh-user=""	如果指定，则通过 SSH 使用指定的用户名对 Node 进行访问
--storage-backend=""	设置持久化存储类型，可选项为 etcd2 (默认)、etcd3
--storage-media-type="application/json"	持久化存储中的保存格式， 默认为 application/json。某些资源类型只能使用 application/json 格式进行保存，将忽略这个参数的设置

续表

参数名和取值示例	说 明
--storage-versions="apps/v1alpha1,authentication.k8s.io/v1beta1,authorization.k8s.io/v1beta1,autoscaling/v1,batch/v1,componentconfig/v1alpha1,extensions/v1beta1,policy/v1alpha1,rbac.authorization.k8s.io/v1alpha1,v1"	持久化存储的资源版本号，以分组形式标记，例如“group1/version1,group2/version2,...”
--tls-cert-file=""	包含 x509 证书的文件路径，用于 HTTPS 认证
--tls-private-key-file=""	包含 x509 与 tls-cert-file 对应的私钥文件路径
--token-auth-file=""	用于访问 API Server 安全端口的 token 认证文件路径
--watch-cache[=true]	设置为 true 表示将 watch 进行缓存
--watch-cache-sizes=[]	设置各资源对象 watch 缓存大小的列表，以逗号分隔，每个资源对象的设置格式为 resource#size，当 watch-cache 设置为 true 时生效

3. kube-controller-manager 启动参数

对 kube-controller-manager 启动参数的详细说明如表 2.5 所示。

表 2.5 对 kube-controller-manager 启动参数的详细说明

参数名和取值示例	说 明
--address=0.0.0.0	监听的主机 IP 地址，默认为 0.0.0.0 表示使用全部网络接口
--allocate-node-cidrs=false	设置为 true 表示使用云服务商为 Pod 分配的 CIDRs，仅用于公有云
--cloud-config=""	云服务商的配置文件路径，仅用于公有云
--cloud-provider=""	云服务商的名称，仅用于公有云
--cluster-cidr=<nil>	集群中 Pod 的可用 CIDR 范围
--cluster-name="kubernetes"	集群的名称，默认为 kubernetes
--concurrent-deployment-syncs=5	设置允许的并发同步 deployment 对象的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-endpoint-syncs=5	设置并发执行 Endpoint 同步操作的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-rc-syncs=5	并发执行 RC 同步操作的协程数，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-namespace-syncs=2	设置允许的并发同步 namespace 对象的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-rc-syncs=5	设置允许的并发同步 replication controller 对象的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源
--concurrent-replicaset-syncs=5	设置允许的并发同步 replica set 对象的数量，值越大表示同步操作越快，但将会消耗更多的 CPU 和网络资源

续表

参数名和取值示例	说 明
--concurrent-resource-quota-syncs=5	设置允许的并发同步 resource quota 对象的数量，值越大表示更快地进行同步操作，但将会消耗更多的 CPU 和网络资源
--configure-cloud-routes[=true]	设置为 true 表示使用 allocate-node-cidrs 进行 CIDRs 的分配，仅用于公有云
--controller-start-interval=0	启动各个 controller manager 的时间间隔，默认为 0 秒
--daemonset-lookup-cache-size=1024	DaemonSet 的查询缓存大小，默认为 1024。值越大表示 DaemonSet 响应越快，内存消耗也越大
--deleting-pods-burst=10	如果一个 Node 节点失败，则会批量删除在上面运行的 Pod 实例的信息，此值定义了突发最大删除的 Pod 的数量，与 deleting-pods-qps 一起作为调度中的限流因子
--deleting-pods-qps=0.1	当 Node 失效时，每秒删除其上的多少个 Pod 实例
--deployment-controller-sync-period=30s	同步 deployments 的时间间隔，默认为 30 秒
--enable-dynamic-provisioning[=true]	设置为 true 表示启用动态 provisioning（需环境支持）
--enable-garbage-collector[=false]	设置为 true 表示启用垃圾回收机制，必须与 kube-apiserver 的该参数设置为相同的值
--enable-hostpath-provisioner[=false]	设置为 true 表示启用 hostPath PV provisioning 机制，仅用于测试，不可用于多 Node 的集群环境
--flex-volume-plugin-dir="/usr/libexec/kubernetes/kubelet-plugins/volume/exec/"	设置 flex volume 插件应搜索其他第三方 volume 插件的全路径
--horizontal-pod-autoscaler-sync-period=30s	Pod 自动扩容器的 Pod 数量的同步时间间隔，默认为 30 秒
--kube-api-burst=30	发送到 API Server 的每秒的请求数量，默认为 30
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=20	与 API Server 通信的 QPS 值，默认为 20
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--leader-elect[=false]	设置为 true 表示进行 leader 选举，用于多个 Master 组件的高可用部署
--leader-elect-lease-duration=15s	leader 选举过程中非 leader 等待选举的时间间隔，默认为 15 秒，当 leader-elect=true 时生效
--leader-elect-renew-deadline=10s	leader 选举过程中在停止 leading 角色之前再次 renew 的时间间隔，应小于或等于 leader-elect-lease-duration， 默认为 10 秒，当 leader-elect=true 时生效
--leader-elect-retry-period=2s	leader 选举过程中在获取 leader 角色和 renew 之间的等待时间， 默认为两秒，当 leader-elect=true 时生效
--master=""	API Server 的 URL 地址，设置后不再使用 kubeconfig 中设置的值
--min-resync-period=12h0m0s	最小重新同步的时间间隔，实际重新同步的时间为 MinResyncPeriod（默认为 12 小时）到 $2 \times \text{MinResyncPeriod}$ （默认 24 小时）之间的一个随机数
--namespace-sync-period=5m0s	namespace 生命周期更新的同步时间间隔， 默认为 5 分钟
--node-cidr-mask-size=24	Node CIDR 的子网掩码设置， 默认为 24

续表

参数名和取值示例	说 明
--node-monitor-grace-period=40s	监控 Node 状态的时间间隔，默认为 40 秒，超过该设置时间后，controller-manager 会把 Node 标记为不可用状态。此值的设置有如下要求： 它应该被设置为 kubelet 汇报的 Node 状态时间间隔（参数—node-status-update-frequency=10s）的 N 倍，N 为 kubelet 状态汇报的重试次数
--node-monitor-period=5s	同步 NodeStatus 的时间间隔，默认为 5 秒
--node-startup-grace-period=1m0s	Node 启动的最大允许时间，超过此时间无响应则会标记 Node 为不可用状态（启动失败），默认为 1 分钟
--node-sync-period=10s	Node 信息发生变化时（例如新 Node 加入集群）controller-manager 同步各 Node 信息的时间间隔，默认为 10 秒
--pod-eviction-timeout=5m0s	在发现一个 Node 失效以后，延迟一段时间，在超过这个参数指定的时间后，删除此 Node 上的 Pod， 默认为 5 分钟
--port=10252	controller-manager 监听的主机口号， 默认为 10252
--profiling=true	打开性能分析，可以通过<host>:<port>/debug/pprof/地址查看程序栈、线程等系统运行信息
--pv-recycler-increment-timeout-nfs=30	使用 nfs scrubber 的 Pod 每增加 1Gi 空间在 ActiveDeadlineSeconds 上增加的时间， 默认为 30 秒
--pv-recycler-minimum-timeout-hostpath=60	使用 hostPath recycler 的 Pod 的最小 ActiveDeadlineSeconds 秒数， 默认为 60 秒。 实验用
--pv-recycler-minimum-timeout-nfs=300	使用 nfs recycler 的 Pod 的最小 ActiveDeadlineSeconds 秒数， 默认为 300 秒
--pv-recycler-pod-template-filename-hostpath=""	使用 hostPath recycler 的 Pod 的模板文件全路径，仅用于实验
--pv-recycler-pod-template-filename-nfs=""	使用 nfs recycler 的 Pod 的模板文件全路径
--pv-recycler-timeout-increment-hostpath=30	使用 hostPath scrubber 的 Pod 每增加 1Gi 空间在 ActiveDeadlineSeconds 上增加的时间， 默认为 30 秒。实验用
--pvclaimbinder-sync-period=15s	同步 PV 和 PVC（容器声明的 PV）的时间间隔
--replicaset-lookup-cache-size=4096	设置 replica sets 查询缓存的大小， 默认为 4096， 值越大表示查询操作越快，但将会消耗更多的内存
--replication-controller-lookup-cache-size=4096	设置 replication controller 查询缓存的大小， 默认为 4096， 值越大表示查询操作越快，但将会消耗更多的内存
--resource-quota-sync-period=5m0s	resource quota 使用信息同步的时间间隔， 默认为 5 分钟
--root-ca-file=""	根 CA 证书文件路径，将被用于 Service Account 的 token secret 中
--service-account-private-key-file=""	用于给 Service Account token 签名的 PEM-encoded RSA 私钥文件路径
--service-cluster-ip-range=""	Service 的 IP 范围
--service-sync-period=5m0s	同步 service 与外部 load balancer 的时间间隔， 默认为 5 分钟
--terminated-pod-gc-threshold=12500	设置可保存的终止 Pod 的数量，超过该数量，垃圾回收器将开始进行删除操作。 设置为不大于 0 的值表示不启用该功能

4. kube-scheduler 启动参数

对 kube-scheduler 启动参数的详细说明如表 2.6 所示。

表 2.6 对 kube-scheduler 启动参数的详细说明

参数名和取值示例	说 明
--address=0.0.0.0	监听的主机 IP 地址，默认为 0.0.0.0 表示使用全部网络接口
--algorithm-provider="DefaultProvider"	设置调度算法，默认为 DefaultProvider
--failure-domains="kubernetes.io/hostname,failure-domain.beta.kubernetes.io/zone,failure-domain.beta.kubernetes.io/region"	表示 Pod 调度时的亲和力参数。在调度 Pod 时，如果两个 Pod 有相同的亲和力参数，那么这两个 Pod 会被调度到相同的 Node 上；如果两个 Pod 有不同的亲和力参数，那么这两个 Pod 不会被调度到相同的 Node 上
--hard-pod-affinity-symmetric-weight=1	表示 Pod 调度规则亲和力的权重值，取值范围为 0~100。RequiredDuringScheduling 亲和性是非对称的，但对每一个 RequiredDuringScheduling 亲和性都存在一个对应的隐式 PreferredDuringScheduling 亲和性规则。该设置表示隐式 PreferredDuringScheduling 亲和性规则的权重值，默认为 1
--kube-api-burst=100	发送到 API Server 的每秒请求数量，默认为 100
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=50	与 API Server 通信的 QPS 值，默认为 50
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 的地址信息及必要的认证信息
--leader-elect[=false]	设置为 true 表示进行 leader 选举，用于多个 Master 组件的高可用部署
--leader-elect-lease-duration=15s	leader 选举过程中非 leader 等待选举的时间间隔，默认为 15 秒，当 leader-elect=true 时生效
--leader-elect-renew-deadline=10s	leader 选举过程中在停止 leading 角色之前再次 renew 的时间间隔，应小于或等于 leader-elect-lease-duration，默认为 10 秒，当 leader-elect=true 时生效
--leader-elect-retry-period=2s	leader 选举过程中获取 leader 角色和 renew 之间的等待时间，默认为两秒，当 leader-elect=true 时生效
--master=""	API Server 的 URL 地址，设置后不再使用 kubeconfig 中设置的值
--policy-config-file=""	调度策略（scheduler policy）配置文件的路径
--port=10251	scheduler 监听的主机端口号，默认为 10251
--profiling=true	打开性能分析，可以通过 <host>:<port>/debug/pprof 地址查看栈、线程等系统运行信息
--scheduler-name="default-scheduler"	调度器名称，用于选择哪些 Pod 将被该调度器进行处理，选择的依据是 Pod 的 annotation 设置，包含 key='scheduler.alpha.kubernetes.io/name' 的 annotation

5. kubelet 启动参数

对 kubelet 启动参数的详细说明如表 2.7 所示。

表 2.7 对 kubelet 启动参数的详细说明

参数名和取值示例	说 明
--address=0.0.0.0	绑定主机 IP 地址， 默认为 0.0.0.0 表示使用全部网络接口
--allow-privileged[=false]	是否允许以特权模式启动容器， 默认为 false
--api-servers=[]	API Server 地址列表， 以 ip:port 格式表示， 以逗号分隔
--application-metrics-count-limit=100	为每个容器保存的性能指标的最大数量， 默认为 100
--boot-id-file="/proc/sys/kernel/random/boot_id"	以逗号分隔的文件列表， 使用第 1 个存在 book-id 的文件
--cadvisor-port=4194	本地 cAdvisor 监听的端口号， 默认为 4194
--cert-dir="/var/run/kubernetes"	TLS 证书所在的目录， 默认为 /var/run/kubernetes。如果设置了 --tls-cert-file 和 --tls-private-key-file，则该设置将被忽略
--cgroup-root=""	为 pods 设置的 root cgroup， 如果不设置，则将使用容器运行时的默认设置
--chaos-chance=0	随机产生客户端错误的概率， 仅用于测试， 默认为 0， 即不产生
--cloud-config=""	云服务商的配置文件路径
--cloud-provider="auto-detect"	云服务商的名称， 默认将自动检测， 设置为空表示无云服务商
--cluster-dns=""	集群内 DNS 服务的 IP 地址
--cluster-domain=""	集群内 DNS 服务所用域名
--config=""	kubelet 配置文件的路径或目录名
--configure-cbr0[=false]	设置为 true 表示 kubelet 将会根据 Node.Spec.PodCIDR 的值来配置 cbr0
--container-hints="/etc/cadvisor/container_hints.json"	容器 hints 文件所在的全路径
--container-runtime="docker"	容器类型， 目前支持 Docker、rkt， 默认为 docker
--containerized[=false]	将 kubelet 运行在容器中， 仅供测试使用， 默认为 false
--cpu-cfs-quota[=true]	设置为 true 表示启用 CPU CFS quota， 用于设置容器的 CPU 限制
--docker-endpoint="unix:///var/run/docker.sock"	Docker 服务的 Endpoint 地址， 默认为 unix:///var/run/docker.sock
--docker-env-metadata-whitelist=""	Docker 容器需要使用的环境变量 key 列表， 以逗号分隔
--docker-exec-handler="native"	进入 Docker 容器中执行命令的方式， 支持 native、nsenter， 默认为 native
--docker-only[=false]	设置为 true， 表示仅报告 Docker 容器的统计信息而不再报告其他统计信息
--docker-root="/var/lib/docker"	Docker 根目录的全路径， 不再使用， 将通过 docker info 获取该信息
--enable-controller-attach-detach[=true]	设置为 true 表示启用 Attach/Detach Controller 进行调度到该 Node 的 volume 的 attach 与 detach 操作， 同时禁用 kubelet 执行 attach、detach 的操作
--enable-custom-metrics[=false]	设置为 true 表示启用采集自定义性能指标
--enable-debugging-handlers=false	设置为 true 表示提供远程访问本节点容器的日志、进入容器执行命令等相关 Rest 服务
--enable-load-reader[=false]	设置为 true 表示启用 CPU 负载的 reader
--enable-server[=true]	启动 kubelet 上的 http rest server， 此 server 提供了获取本节点上运行的 Pod 列表、Pod 状态和其他管理监控相关的 Rest 接口

续表

参数名和取值示例	说 明
--event-burst=10	临时允许的 Event 记录突发的最大数量，默认为 10，当 event-qps>0 时生效
--event-qps=5	设置大于 0 的值表示限制每秒能创建出的 Event 数量，设置为 0 表示不限制
--event-storage-age-limit="default=0"	保存 Event 的最大时间。按事件类型以 key=value 的格式表示，以逗号分隔，事件类型包括 creation、oom 等，“default” 表示所有事件的类型
--event-storage-event-limit="default=0"	保存 Event 的最大数量。按事件类型以 key=value 格式表示，以逗号分隔，事件类型包括 creation、oom 等，“default” 表示所有事件的类型
--eviction-hard=""	触发 Pod Eviction 操作的一组硬门限设置，例如可用内存<1Gi
--eviction-max-pod-grace-period=0	终止 Pod 操作给 Pod 自行停止预留的时间，单位为秒。时间到达时，将触发 Pod Eviction 操作。默认值为 0，设置为负数表示使用 Pod 中指定的值
--eviction-pressure-transition-period=5m0s	kubelet 在触发 Pod Eviction 操作之前等待的最长时间，默认为 5 分钟
--eviction-soft=""	触发 Pod Eviction 操作的一组软门限设置，例如可用内存<1.5Gi，与 grace-period 一起生效，当 Pod 的响应时间超过 grace-period 后进行触发
--eviction-soft-grace-period=""	触发 Pod Eviction 操作的一组软门限等待时间设置，例如 memory.available=1m30s
--exit-on-lock-contention[=false]	设置为 true 表示当有文件锁存在时 kubelet 也可以退出
--experimental-flannel-overlay[=false]	实验性功能，用于 kubelet 启动时自动支持 flannel 覆盖网络，默认值为 false
--experimental-nvidia-gpus=0	本节点上 NVIDIA GPU 的数量，目前仅支持 0 或 1，默认为 0
--file-check-frequency=20s	在 File Source 作为 Pod 源的情况下，kubelet 定期重新检查文件变化的时间间隔，文件发生变化后，kubelet 重新加载更新的文件内容
--global-housekeeping-interval=1m0s	全局 housekeeping 的时间间隔，默认为 1 分钟
--google-json-key=""	用于谷歌的云平台 Service Account 进行用于鉴权的 JSON key
--hairpin-mode="promiscuous-bridge"	设置 hairpin 模式，表示 kubelet 设置 hairpin NAT 的方式。该模式允许后端 Endpoint 在访问其本身 Service 时能够再次 loadbalance 回自身。可选项包括 promiscuous-bridge、hairpin-veth 和 none
--healthz-bind-address=127.0.0.1	healthz 服务监听的 IP 地址，默认为 127.0.0.1，设置为 0.0.0.0 表示监听全部 IP 地址
--healthz-port=10248	本地 healthz 服务监听的端口号，默认为 10248
--host-ipc-sources="*"	kubelet 允许 Pod 使用宿主机 ipc namespace 的列表，以逗号分隔，默认为 “*”
--host-network-sources="*"	kubelet 允许 Pod 使用宿主机 network 的列表，以逗号分隔，默认为 “*”
--host-pid-sources="*"	kubelet 允许 Pod 使用宿主机 pid namespace 的列表，以逗号分隔，默认为 “*”
--hostname-override=""	设置本 Node 在集群中的主机名，不设置将使用本机 hostname
--housekeeping-interval=10s	对容器做 housekeeping 操作的时间间隔，默认为 10 秒
--http-check-frequency=20s	HTTP URL Source 作为 Pod 源的情况下，kubelet 定期检查 URL 返回的内容是否发生变化的时间周期，作用同 file-check-frequency 参数
--image-gc-high-threshold=90	镜像垃圾回收上限，磁盘使用空间达到该百分比时，镜像垃圾回收将持续工作
--image-gc-low-threshold=80	镜像垃圾回收下限，磁盘使用空间在达到该百分比之前，镜像垃圾回收将不启用
--kube-api-burst=10	发送到 API Server 的每秒请求数量，默认为 10

续表

参数名和取值示例	说 明
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=5	与 API Server 通信的 QPS 值, 默认为 5
--kube-reserved=	kubernetes 系统预留的资源配置, 以一组 resourceName=resourceQuantity 格式表示, 例如 cpu=200m, memory=150G。目前仅支持 CPU 和内存的设置, 详见 http://releases.k8s.io/HEAD/docs/user-guide/compute-resources.md , 默认为空
--kubeconfig="/var/lib/kubelet/kubeconfig"	kubeconfig 配置文件路径, 在配置文件中包括 Master 地址信息及必要的认证信息
--kublet-cgroups=""	kubelet 运行所需的 cgroups 名称
--lock-file=""	kubelet 使用的 lock 文件, Alpha 版本
--log-cadvisor-usage[=false]	设置为 true 表示将 cAdvisor 容器的使用情况进行日志记录
--low-diskspace-threshold-mb=256	本 Node 最低磁盘可用空间, 单位 MB。当磁盘空间低于该阈值, kubelet 将拒绝创建新的 Pod, 默认值为 256MB
--machine-id-file="/etc/machine-id,/var/lib/dbus/machine-id"	用于查找 machine-id 的文件列表, 使用找到的第 1 个值, 默认从 /etc/machine-id, /var/lib/dbus/machine-id 文件中去查找
--manifest-url=""	为 HTTP URL Source 源类型时, kubelet 用来获取 Pod 定义的 URL 地址, 此 URL 返回一组 Pod 定义
--manifest-url-header=""	访问 manifest URL 地址时使用的 HTTP 头信息, 以 key:value 格式表示
--master-service-namespace="default"	Master 服务的命名空间, 默认为 default
--max-open-files=1000000	kubelet 打开的最大文件数量, 默认为 1000 000
--max-pods=110	kubelet 能运行的最大 Pod 数量, 默认为 110 个 Pod
--maximum-dead-containers=240	在本 Node 上保留的已停止容器的最大数量, 由于停止的容器也会消耗磁盘空间, 所以超过该上限以后, kubelet 会自动清理已停止的容器以释放磁盘空间, 默认为 240
--maximum-dead-containers-per-container=2	以 Pod 为单位可以保留的已停止的(属于同一 Pod 的)容器集的最大数量
--minimum-container-ttl-duration=1m0s	已停止的容器在被清理之前的最小存活时间, 例如 300ms、10s 或 2h45m, 超过此存活时间的容器将被标记为可被 GC 清理, 默认值为 1 分钟
--minimum-image-ttl-duration=2m0s	不再使用的镜像在被清理之前的最小存活时间, 例如 300ms、10s 或 2h45m, 超过此存活时间的镜像被标记为可被 GC 清理, 默认值为两分钟
--network-plugin=""	自定义的网络插件的名字, Pod 的生命周期中相关的一些事件会调用此网络插件进行处理, 为 Alpha 测试版功能
--network-plugin-dir="/usr/libexec/kubernetes/kubelet-plugins/net/exec/"	扫描网络插件的目录, 为 Alpha 测试版功能
--node-ip=""	设置本 Node 的 IP 地址
--node-labels=	kubelet 注册本 Node 时设置的 Labels, label 以 key=value 的格式表示, 多个 label 以逗号分隔, 为 Alpha 测试版功能
--node-status-update-frequency=10s	kubelet 向 Master 汇报 Node 状态的时间间隔, 默认值为 10 秒。与 controller-manager 的--node-monitor-grace-period 参数共同起作用
--non-masquerade-cidr="10.0.0.0/8"	kubelet 向该 IP 段之外的 IP 地址发送的流量将使用 IP Masquerade 技术

续表

参数名和取值示例	说 明
--oom-score-adj=-999	kubelet 进程的 oom_score_adj 参数值，有效范围为[-1000, 1000]
--outoffdisk-transition-frequency=5m0s	触发磁盘空间耗尽操作之前的等待时间，默认为 5 分钟
--pod-cidr=""	用于给 Pod 分配 IP 地址的 CIDR 地址池，仅在单机模式中使用。在一个集群中，kubelet 会从 API Server 中获取 CIDR 设置
--pod-infra-container-image="gcr.io/google_containers/pause-amd64:3.0"	用于 Pod 内网络命名空间共享的基础 pause 镜像
--pods-per-core=0	该 kubelet 上每个 core 可运行的 Pod 数量。最大值将被 max-pods 参数限制。默认值为 0 表示不做限制
--port=10250	kubelet 服务监听的本机端口号，默认为 10250
--read-only-port=10255	kubelet 服务监听的“只读”端口号， 默认为 10255，设置为 0 表示不启用
--really-crash-for-testing=false	设置为 true 表示发生 panics 情况时崩溃，仅用于测试
--reconcile-cidr[=true]	根据 API Server 指定的 CIDR 重排 Node 的 CIDR 地址，如果 register-node 或 configure-cbr0 设置为 false，则表示不启用。默认值为 true
--register-node[=true]	将本 Node 注册到 API Server， 默认值为 true
--register-schedulable[=true]	将本 Node 状态标记为 schedulable，设置为 false 表示通知 Master 本 Node 不可进行调度。默认值为 true
--registry-burst=10	最多同时拉取镜像的数量， 默认值为 10
--registry-qps=5	在 Pod 创建过程中容器的镜像可能需要从 Registry 中拉取，由于拉取镜像的过程中会消耗大量带宽，因此可能需要限速，此参数与 registry-burst 一起用来限制每秒拉取多少个镜像， 默认不限速，如果设置为 5，则表示平均每秒允许拉取 5 个镜像
--resolv-conf="/etc/resolv.conf"	命名服务配置文件，用于容器内应用的 DNS 解析， 默认为 /etc/resolv.conf
--rkt-api-endpoint="localhost:15441"	rkt API 服务的 URL 地址，--container-runtime='rkt' 时生效
--rkt-path=""	rkt 二进制文件的路径，不指定的话从环境变量\$PATH 中查找，--container-runtime='rkt' 时生效
--root-dir="/var/lib/kubelet"	kubelet 运行根目录，将保持 Pod 和 volume 的相关文件， 默认为 /var/lib/kubelet。
--runonce=false	设置为 true 表示创建完 Pod 之后立即退出 kubelet 进程，与 --api-servers 和 --enable-server 参数互斥
--runtime-cgroups=""	为容器 runtime 设置的 cgroup
--runtime-request-timeout=2m0s	除了长时间运行的 request，对其他 request 的超时时间设置，包括 pull、logs、exec、attach 等操作。当超时时间到达时，请求会被杀掉，抛出一个错误并会重试。默认值为两分钟
--seccomp-profile-root="/var/lib/kubelet/seccomp"	seccomp 配置文件目录， 默认为 /var/lib/kubelet/seccomp
--serialize-image-pulls[=true]	按顺序挨个 pull 镜像。建议 Docker 低于<1.9 版本或使用 Aufs storage backend 时设置为 true，详见 issue #10959
--storage-driver-buffer-duration=1m0s	将缓存数据写入后端存储的时间间隔， 默认为 1 分钟

续表

参数名和取值示例	说 明
--storage-driver-db="cadvisor"	后端存储的数据库名称, 默认为 cadvisor
--storage-driver-host="localhost:8086"	后端存储的数据库连接 URL 地址, 默认为 localhost:8086
--storage-driver-password="root"	后端存储的数据库密码, 默认为 root
--storage-driver-secure[=false]	后端存储的数据库是否用安全连接, 默认为 false
--storage-driver-table="stats"	后端存储的数据库表名, 默认为 stats
--storage-driver-user="root"	后端存储的数据库用户名, 默认为 root
--streaming-connection-idle-timeout=4h0m0s	在容器中执行命令或者进行端口转发的过程中会产生输入、输出流, 这个参数用来控制连接空闲超时而关闭的时间, 如果设置为“5m”, 则表示连接超过 5 分钟没有输入、输出的情况下就被认为是空闲的, 而会被自动关闭。默认为 4 小时
--sync-frequency=1m0s	同步运行中容器的配置的频率, 默认为 1 分钟
--system-cgroups=""	kubelet 为运行非 kernel 进程设置的 cgroups 名称
--system-reserved=	系统预留的资源配置, 以一组 ResourceName=ResourceQuantity 格式表示, 例如 cpu=200m, memory=150G。目前仅支持 CPU 和内存的设置, 详见 http://releases.k8s.io/HEAD/docs/user-guide/compute-resources.md , 默认为空
--tls-cert-file=""	包含 x509 证书的文件路径, 用于 HTTPS 认证
--tls-private-key-file=""	包含 x509 与 tls-cert-file 对应的私钥文件路径
--volume-plugin-dir="/usr/libexec/kubernetes/kubelet-plugins/volume/exec/"	搜索第三方 volume 插件的目录, 为 Alpha 测试版功能
--volume-stats-agg-period=1m0s	kubelet 计算所有 Pod 和 volume 的磁盘使用情况聚合值的时间间隔, 默认为 1 分钟。设置为 0 表示不启用该计算功能

6. kube-proxy 启动参数

`kube-proxy` 的启动参数详细说明见表 2.8。

表 2.8 `kube-proxy` 的参数表

参数名和取值示例	说 明
--bind-address=0.0.0.0	<code>kube-proxy</code> 绑定主机的 IP 地址, 默认为 0.0.0.0 表示绑定所有 IP 地址
--cleanup-iptables[=false]	设置为 true 表示清除 iptables 规则后退出
--cluster-cidr=""	集群中 Pod 的 CIDR 地址范围, 用于桥接集群外部流量到内部。用于公有云环境
--config-sync-period=15m0s	从 API Server 更新配置的时间间隔, 默认为 15 分钟, 必须大于 0
--conntrack-max=0	跟踪 NAT 连接的最大数量, 默认值为 0 表示不限制
--conntrack-max-per-core=32768	跟踪每个 CPU core 的 NAT 连接的最大数量, 默认值为 32768, 仅当 conntrack-max 设置为 0 时生效
--conntrack-tcp-timeout-established=24h0m0s	建立 TCP 连接的超时时间, 默认为 24 小时, 设置为 0 表示无限制

续表

参数名和取值示例	说 明
--healthz-bind-address=127.0.0.1	healthz 服务绑定主机 IP 地址，默认为 127.0.0.1，设置为 0.0.0.0 表示使用所有 IP 地址
--healthz-port=10249	healthz 服务监听的主机端口号，默认为 10249
--hostname-override=""	设置本 Node 在集群中的主机名，不设置将使用本机 hostname
--iptables-masquerade-bit=14	iptables masquerade 的 fwmark 位设置，有效范围为[0, 31]
--iptables-sync-period=30s	刷新 iptables 规则的时间间隔，例如 5s、1m、2h22m，默认为 30 秒，必须大于 0
--kube-api-burst=10	发送到 API Server 的每秒发请求数量，默认为 10
--kube-api-content-type="application/vnd.kubernetes.protobuf"	发送到 API Server 的请求内容类型
--kube-api-qps=5	与 API Server 通信的 QPS 值，默认为 5
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--masquerade-all[=false]	设置为 true 表示使用纯 iptables 代理，所有网络包都将做 SNAT 转换
--master=""	API Server 的地址
--oom-score-adj=-999	kube-proxy 进程的 oom_score_adj 参数值，有效范围为[-1000,1000]
--proxy-mode=	代理模式，可选项为 iptables 或 userspace，默认为 iptables，转发速度更快。当操作系统 kernel 版本或 iptables 版本不够新时，将自动降级为 userspace 模式
--proxy-port-range=	进行 Service 代理的本地端口号范围，格式为 begin-end，含两端，未指定则采用随机选择的系统可用的端口号
--udp-timeout=250ms	保持空闲 UDP 连接的时间，例如 250ms、2s，默认值为 250ms，必须大于 0，仅当 proxy-mode=userspace 时生效

2.1.7 Kubernetes 集群网络配置方案

在多个 Node 组成的 Kubernetes 集群内，跨主机的容器间网络互通是 Kubernetes 集群能够正常工作的前提条件。Kubernetes 本身并不会对跨主机容器网络进行设置，这需要额外的工具来实现。除了谷歌公有云 GCE 平台提供的网络设置，一些开源的工具包括 flannel、Open vSwitch、Weave、Calico 等都能够实现跨主机的容器间网络互通。本节将对常用的 flannel、Open vSwitch 和直接路由三种配置进行详细说明。

1. flannel（覆盖网络）

flannel 采用覆盖网络（Overlay Network）模型来完成对网络的打通，本节对 flannel 的安装和配置进行详细说明。

1) 安装 etcd

由于 flannel 使用 etcd 作为数据库，所以需要预先安装好 etcd，详见 2.1.2 节的说明。

2) 安装 flannel

需要在每台 Node 上都安装 flannel。flannel 软件的下载地址为 <https://github.com/coreos/flannel/releases>。将下载的压缩包 flannel-<version>-linux-amd64.tar.gz 解压，把二进制文件 flanneld 和 mk-docker-opt.sh 复制到 /usr/bin（或其他 PATH 环境变量中的目录），即可完成对 flannel 的安装。

3) 配置 flannel

此处以使用 systemd 系统为例对 flanneld 服务进行配置。编辑服务配置文件 /usr/lib/systemd/system/flanneld.service：

```
[Unit]
Description=flanneld overlay address etcd agent
After=network.target
Before=docker.service

[Service]
Type=notify
EnvironmentFile=/etc/sysconfig/flanneld
ExecStart=/usr/bin/flanneld -etcd-endpoints=${FLANNEL_ETCD} ${FLANNEL_OPTIONS}

[Install]
RequiredBy=docker.service
WantedBy=multi-user.target
```

编辑配置文件 /etc/sysconfig/flannel，设置 etcd 的 URL 地址：

```
# flanneld configuration options

# etcd url location. Point this to the server where etcd runs
FLANNEL_ETCD="http://192.168.18.3:2379"

# etcd config key. This is the configuration key that flannel queries
# For address range assignment
FLANNEL_ETCD_KEY="/coreos.com/network"
```

在启动 flanneld 服务之前，需要在 etcd 中添加一条网络配置记录，这个配置将用于 flanneld 分配给每个 Docker 的虚拟 IP 地址段。

```
# etcdctl set /coreos.com/network/config '{ "Network": "10.1.0.0/16" }'
```

4) 由于 flannel 将覆盖 docker0 网桥，所以如果 Docker 服务已启动，则停止 Docker 服务。

5) 启动 flanneld 服务：

```
# systemctl restart flanneld
```

6) 设置 docker0 网桥的 IP 地址：

```
# mk-docker-opts.sh -i  
# source /run/flannel/subnet.env  
# ifconfig docker0 ${FLANNEL_SUBNET}
```

完成后确认网络接口 docker0 的 IP 地址属于 flannel0 的子网：

```
# ip addr  
flannel0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST> mtu 1472  
    inet 10.1.10.0 netmask 255.255.0.0 destination 10.1.10.0  
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
    inet 10.1.10.1 netmask 255.255.255.0 broadcast 10.1.10.255
```

7) 重新启动 Docker 服务：

```
# systemctl restart docker
```

到此就完成了 flannel 覆盖网络的设置。

使用 ping 命令验证各 Node 上 docker0 之间的相互访问。例如在 Node1(docker0 IP=10.1.10.1) 机器上 ping Node2 的 docker0 (docker0's IP=10.1.30.1)，通过 flannel 能够成功连接到其他物理机的 Docker 网络：

```
$ ping 10.1.30.1  
PING 10.1.30.1 (10.1.30.1) 56(84) bytes of data.  
64 bytes from 10.1.30.1: icmp_seq=1 ttl=62 time=1.15 ms  
64 bytes from 10.1.30.1: icmp_seq=2 ttl=62 time=1.16 ms  
64 bytes from 10.1.30.1: icmp_seq=3 ttl=62 time=1.57 ms
```

我们也可以在 etcd 中查看到 flannel 设置的 flannel0 地址与物理机 IP 地址的对应规则：

```
# etcdctl ls /coreos.com/network/subnets  
/coreos.com/network/subnets/10.1.10.0-24  
/coreos.com/network/subnets/10.1.20.0-24  
/coreos.com/network/subnets/10.1.30.0-24  
  
# etcdctl get /coreos.com/network/subnets/10.1.10.0-24  
{"PublicIP": "192.168.1.129"}  
# etcdctl get /coreos.com/network/subnets/10.1.20.0-24  
{"PublicIP": "192.168.1.130"}  
# etcdctl get /coreos.com/network/subnets/10.1.30.0-24  
{"PublicIP": "192.168.1.131"}
```

2. Open vSwitch (虚拟交换机)

以两个 Node 为例，目标网络拓扑如图 2.2 所示。

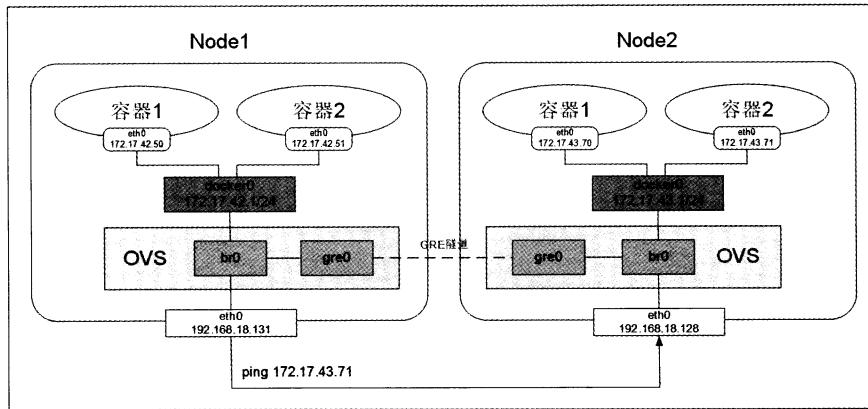


图 2.2 通过 Open vSwitch 打通网络

首先，确保节点 192.168.18.128 的 Docker0 采用 172.17.43.0/24 网段，而 192.168.18.131 的 Docker0 采用 172.17.42.0/24 网段，对应参数为 docker daemon 的启动参数--bip 设置的值。

Open vSwitch 的安装和配置方法如下。

1) 在两个 Node 上安装 ovs

```
# yum install openvswitch-2.4.0-1.x86_64.rpm
```

禁止 selinux，配置后重启 Linux：

```
# vi /etc/selinux/config
SELINUX=disabled
```

查看 Open vSwitch 的服务状态，应该启动两个进程：ovsdb-server 与 ovs-vswitchd。

```
# service openvswitch status
ovsdb-server is running with pid 2429
ovs-vswitchd is running with pid 2439
```

查看 Open vSwitch 的相关日志，确认没有异常：

```
# more /var/log/messages |grep openv
Nov 2 03:12:52 docker128 openvswitch: Starting ovsdb-server [ OK ]
Nov 2 03:12:52 docker128 openvswitch: Configuring Open vSwitch system IDs
[ OK ]
Nov 2 03:12:52 docker128 kernel: openvswitch: Open vSwitch switching datapath
Nov 2 03:12:52 docker128 openvswitch: Inserting openvswitch module [ OK ]
```

注意上述操作需要在两个节点机器上分别执行完成。

2) 创建网桥和 GRE 隧道

接下来需要在每个 Node 上建立 ovs 的网桥 br0，然后在网桥上创建一个 GRE 隧道连接对端网桥，最后把 ovs 的网桥 br0 作为一个端口连接到 docker0 这个 Linux 网桥上（可以认为是交

换机互联），这样一来，两个节点机器上的 docker0 网段就能互通了。

下面以节点机器 192.168.18.131 为例，具体的操作步骤如下。

(1) 创建 ovs 网桥：

```
# ovs-vsctl add-br br0
```

(2) 创建 GRE 隧道连接对端，remote_ip 为对端 eth0 的网卡地址：

```
# ovs-vsctl add-port br0 gre1 -- set interface gre1 type=gre  
option:remote_ip=192.168.18.128
```

(3) 添加 br0 到本地 docker0，使得容器流量通过 OVS 流经 tunnel：

```
# brctl addif docker0 br0
```

(4) 启动 br0 与 docker0 网桥：

```
# ip link set dev br0 up  
# ip link set dev docker0 up
```

(5) 添加路由规则。由于 192.168.18.128 与 192.168.18.131 的 docker0 网段分别为 172.17.43.0/24 与 172.17.42.0/24，这两个网段的路由都需要经过本机的 docker0 网桥路由，其中一个 24 网段是通过 OVS 的 GRE 隧道到达对端的，因此需要在每个 Node 上添加通过 docker0 网桥转发的 172.17.0.0/16 段的路由规则：

```
# ip route add 172.17.0.0/16 dev docker0
```

(6) 清空 Docker 自带的 Iptables 规则及 Linux 的规则，后者存在拒绝 icmp 报文通过防火墙的规则：

```
# iptables -t nat -F; iptables -F
```

在 192.168.18.131 上完成上述步骤后，在 192.168.18.128 节点执行同样的操作，注意，GRE 隧道里的 IP 地址要改为对端节点（192.168.18.131）的 IP 地址。

配置完成后，192.168.18.131 的 IP 地址、docker0 的 IP 地址及路由等重要信息显示如下：

```
# ip addr  
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN  
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  
    inet 127.0.0.1/8 scope host lo  
        valid_lft forever preferred_lft forever  
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP  
    qlen 1000  
    link/ether 00:0c:29:55:5e:c3 brd ff:ff:ff:ff:ff:ff  
    inet 192.168.18.131/24 brd 192.168.18.255 scope global dynamic eth0  
        valid_lft 1369sec preferred_lft 1369sec  
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN  
    link/ether a6:15:c3:25:cf:33 brd ff:ff:ff:ff:ff:ff  
4: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
```

```

state UNKNOWN
    link/ether 92:8d:d0:a4:ca:45 brd ff:ff:ff:ff:ff:ff
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
        link/ether 02:42:44:8d:62:11 brd ff:ff:ff:ff:ff:ff
        inet 172.17.42.1/24 scope global docker0
            valid_lft forever preferred_lft forever

```

同样，192.168.18.128 节点的重要信息如下：

```

# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
qlen 1000
    link/ether 00:0c:29:e8:02:c7 brd ff:ff:ff:ff:ff:ff
    inet 192.168.18.128/24 brd 192.168.18.255 scope global dynamic eth0
        valid_lft 1356sec preferred_lft 1356sec
3: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
    link/ether fa:6c:89:a2:f2:01 brd ff:ff:ff:ff:ff:ff
4: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
state UNKNOWN
    link/ether ba:89:14:e0:7f:43 brd ff:ff:ff:ff:ff:ff
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:63:a8:14:d5 brd ff:ff:ff:ff:ff:ff
    inet 172.17.43.1/24 scope global docker0
        valid_lft forever preferred_lft forever

```

3) 两个 Node 上容器之间的互通测试

首先，在192.168.18.128 节点上 ping 192.168.18.131 上的 docker0 地址：172.17.42.1，验证网络互通性：

```

# ping 172.17.42.1
PING 172.17.42.1 (172.17.42.1) 56(84) bytes of data.
64 bytes from 172.17.42.1: icmp_seq=1 ttl=64 time=1.57 ms
64 bytes from 172.17.42.1: icmp_seq=2 ttl=64 time=0.966 ms
64 bytes from 172.17.42.1: icmp_seq=3 ttl=64 time=1.01 ms
64 bytes from 172.17.42.1: icmp_seq=4 ttl=64 time=1.00 ms
64 bytes from 172.17.42.1: icmp_seq=5 ttl=64 time=1.22 ms
64 bytes from 172.17.42.1: icmp_seq=6 ttl=64 time=0.996 ms

```

下面我们通过 tshark 抓包工具来分析流量走向。首先，在192.168.18.128 节点上监听 br0 上是否有 GRE 报文，执行下面的命令，我们发现 br0 上并没有 GRE 报文：

```

# tshark -i br0 -R ip proto GRE
tshark: -R without -2 is deprecated. For single-pass filtering use -Y.
Running as user "root" and group "root". This could be dangerous.

```

```
Capturing on 'br0'
^C
```

而在 eth0 上抓包，则发现了 GRE 封装的 ping 包报文通过，说明 GRE 是在承载网的物理网上完成的封包过程：

```
# tshark -i eth0 -R ip proto GRE
tshark: -R without -2 is deprecated. For single-pass filtering use -Y.
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
1  0.000000  172.17.43.1 -> 172.17.42.1  ICMP 136 Echo (ping) request
id=0x0970, seq=180/46080, ttl=64
2  0.000892  172.17.42.1 -> 172.17.43.1  ICMP 136 Echo (ping) reply
id=0x0970, seq=180/46080, ttl=64 (request in 1)
2  3  1.002014  172.17.43.1 -> 172.17.42.1  ICMP 136 Echo (ping) request
id=0x0970, seq=181/46336, ttl=64
4  1.002916  172.17.42.1 -> 172.17.43.1  ICMP 136 Echo (ping) reply
id=0x0970, seq=181/46336, ttl=64 (request in 3)
4  5  2.004101  172.17.43.1 -> 172.17.42.1  ICMP 136 Echo (ping) request
id=0x0970, seq=182/46592, ttl=64
```

至此，基于 OVS 的网络搭建成功，由于 GRE 是点对点隧道通信方式，所以如果有多个 Node，则需要建立 $N \times (N-1)$ 条 GRE 隧道，即所有 Node 组成一个网状的网络，实现全网互通。

3. 直接路由

通过在每个 Node 上添加到其他 Node 上 docker0 的静态路由规则，就可以将不同物理机的 docker0 网桥互联互通。图 2.3 描述了在两个 Node 之间打通网络的情况。

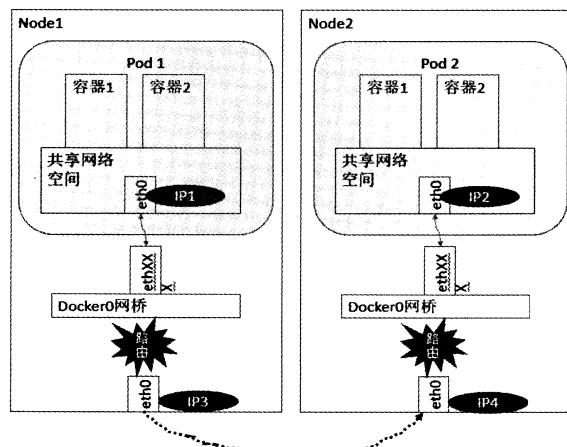


图 2.3 以直接路由方式实现 Pod 到 Pod 的通信

使用这种方案，只需要在每个 Node 的路由表中增加到对方 docker0 的静态路由转发规则。

例如 Pod1 所在 docker0 网桥的 IP 子网是 10.1.10.0，Node 地址为 192.168.1.128；而 Pod2 所在 docker0 网桥的 IP 子网是 10.1.20.0，Node 地址为 192.168.1.129。

在 Node1 上用 route add 命令增加一条到 Node2 上 docker0 的静态路由规则：

```
route add -net 10.1.20.0 netmask 255.255.255.0 gw 192.168.1.129
```

同样，在 Node2 上增加一条到 Node1 上 docker0 的静态路由规则：

```
route add -net 10.1.10.0 netmask 255.255.255.0 gw 192.168.1.128
```

在 Node1 上通过 ping 命令验证到 Node2 上 docker0 的网络连通性。这里 10.1.20.1 为 Node2 上 docker0 网桥自身的 IP 地址。

```
$ ping 10.1.20.1
PING 10.1.20.1 (10.1.20.1) 56(84) bytes of data.
64 bytes from 10.1.20.1: icmp_seq=1 ttl=62 time=1.15 ms
64 bytes from 10.1.20.1: icmp_seq=2 ttl=62 time=1.16 ms
64 bytes from 10.1.20.1: icmp_seq=3 ttl=62 time=1.57 ms
.....
```

可以看到，路由转发规则生效，Node1 可以直接访问到 Node2 上的 docker0 网桥，进一步也可以访问到属于 docker0 网段的容器应用了。

不过，集群中机器的数量通常可能很多。假设有 100 台服务器，那么就需要在每台服务器上手工添加到另外 99 台服务器 docker0 的路由规则。为了减少手工操作，可以使用 Quagga 软件来实现路由规则的动态添加。Quagga 软件的主页为 <http://www.quagga.net>。

除了在每台服务器上安装 Quagga 软件并启动，还可以使用 Quagga 容器来运行（例如 index.alauda.cn/georce/router）。在每台 Node 上下载该 Docker 镜像：

```
$ docker pull index.alauda.cn/georce/router
```

在运行 Quagga 容器之前，需要确保每个 Node 上 docker0 网桥的子网地址不能重叠，也不能与物理机所在的网络重叠，这需要网络管理员的仔细规划。

下面以 3 个 Node 为例，每台 Node 的 docker0 网桥的地址如下（前提是 Node 物理机的 IP 地址不是 10.1.X.X 地址段）：

```
Node 1: # ifconfig docker0 10.1.10.1/24
Node 2: # ifconfig docker0 10.1.20.1/24
Node 3: # ifconfig docker0 10.1.30.1/24
```

然后在每个 Node 上启动 Quagga 容器。需要说明的是，Quagga 需要以--privileged 特权模式运行，并且指定--net=host，表示直接使用物理机的网络：

```
$ docker run -itd --name=router --privileged --net=host
index.alauda.cn/georce/router
```

启动成功后，Quagga 会相互学习来完成到其他机器的 docker0 路由规则的添加。

一段时间后，在 Node1 上使用 route -n 命令来查看路由表，可以看到 Quagga 自动添加了两条到 Node2 和到 Node3 上 docker0 的路由规则。

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask        Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.128  0.0.0.0        UG      0      0      0 eth0
10.1.10.0       0.0.0.0        255.255.255.0   U       0      0      0 docker0
10.1.20.0       192.168.1.129  255.255.255.0   UG      20     0      0 eth0
10.1.30.0       192.168.1.130  255.255.255.0   UG      20     0      0 eth0
```

在 Node2 上查看路由表，可以看到自动添加了两条到 Node1 和 Node3 上 docker0 的路由规则。

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask        Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.129  0.0.0.0        UG      0      0      0 eth0
10.1.20.0       0.0.0.0        255.255.255.0   U       0      0      0 docker0
10.1.10.0       192.168.1.128  255.255.255.0   UG      20     0      0 eth0
10.1.30.0       192.168.1.130  255.255.255.0   UG      20     0      0 eth0
```

至此，所有 Node 上的 docker0 都可以互联互通了。

2.2 kubectl 命令行工具用法详解

kubectl 作为客户端 CLI 工具，可以让用户通过命令行的方式对 Kubernetes 集群进行操作。本节对 kubectl 的子命令和用法进行详细说明。

2.2.1 kubectl 用法概述

kubectl 命令行的语法如下：

```
$ kubectl [command] [TYPE] [NAME] [flags]
```

其中，command、TYPE、NAME、flags 的含义如下。

(1) command：子命令，用于操作 Kubernetes 集群资源对象的命令，例如 create、delete、describe、get、apply 等。

(2) TYPE：资源对象的类型，区分大小写，能以单数形式、复数形式或者简写形式表示。例如以下 3 种 TYPE 是等价的。

```
$ kubectl get pod pod1
```

```
$ kubectl get pods pod1
$ kubectl get po pod1
```

(3) NAME: 资源对象的名称, 区分大小写。如果不指定名称, 则系统将返回属于 TYPE 的全部对象的列表, 例如\$ kubectl get pods 将返回所有 Pod 的列表。

(4) flags: kubectl 子命令的可选参数, 例如使用 “-s” 指定 apiserver 的 URL 地址而不用默认值。

kubectl 可操作的资源对象类型如表 2.9 所示。

表 2.9 kubectl 可操作的资源对象类型

资源对象的名称	缩写
componentstatuses	cs
daemonsets	ds
deployments	
events	ev
endpoints	ep
horizontalpodautoscalers	hpa
ingresses	ing
jobs	
limitranges	limits
nodes	no
namespaces	ns
pods	po
persistentvolumes	pv
persistentvolumeclaims	pvc
resourcequotas	quota
replicationcontrollers	rc
secrets	
serviceaccounts	
services	svc

在一个命令行中也可以同时对多个资源对象进行操作, 以多个 TYPE 和 NAME 的组合表示, 示例如下。

◎ 获取多个 Pod 的信息:

```
$ kubectl get pods pod1 pod2
```

◎ 获取多种对象的信息:

```
$ kubectl get pod/pod1 rc/rc1
```

◎ 同时应用多个 yaml 文件, 以多个-f file 参数表示:

```
$ kubectl get pod -f pod1.yaml -f pod2.yaml
$ kubectl create -f pod1.yaml -f rcl.yaml -f servicel.yaml
```

2.2.2 kubectl 子命令详解

kubectl 的子命令非常丰富，涵盖了对 Kubernetes 集群的主要操作，包括资源对象的创建、删除、查看、修改、配置、运行等。详细的子命令如表 2.10 所示。

表 2.10 kubectl 子命令详解

子命令	语法	说明
annotate	kubectl annotate [-ooverwrite] (-f FILENAME TYPE NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--resource-version=version] [flags]	添加或更新资源对象的 annotation 信息
api-versions	kubectl api-versions [flags]	列出当前系统支持的 API 版本列表，格式为“group/version”
apply	kubectl apply -f FILENAME [flags]	从配置文件或 stdin 中对资源对象进行配置更新
attach	kubectl attach POD -c CONTAINER [flags]	附着到一个正在运行的容器上
autoscale	kubectl autoscale (-f FILENAME TYPE NAME TYPE/NAME) [-min=MINPODS] --max=MAXPODS [--cpu-percent=CPU] [flags]	对 Deployment、ReplicaSet 或 ReplicationController 进行水平自动扩缩容的设置
cluster-info	kubectl cluster-info [flags] kubectl cluster-info [command]	显示集群信息
completion	kubectl completion SHELL [flags]	输出 shell 命令的执行结果码（bash 或 zsh）
config	kubectl config SUBCOMMAND [flags] kubectl config [command]	修改 kubeconfig 文件
convert	kubectl convert -f FILENAME [flags]	转换配置文件为不同的 API 版本
cordon	kubectl cordon NODE [flags]	将 Node 标记为 unschedulable，即“隔离”出集群调度范围
create	kubectl create -f FILENAME [flags] kubectl create [command]	从配置文件或 stdin 中创建资源对象
delete	kubectl delete (-f FILENAME TYPE [(NAME -l label --all)]) [flags]	根据配置文件、stdin、资源名称或 label selector 删除资源对象
describe	kubectl describe (-f FILENAME TYPE [NAME_PREFIX /NAME -l label]) [flags]	描述一个或多个资源对象的详细信息
drain	kubectl drain NODE [flags]	首先将 Node 设置为 unschedulable，然后删除该 Node 上运行的所有 Pod，但不会删除不由 apiserver 管理的 Pod
edit	kubectl edit (RESOURCE/NAME -f FILENAME) [flags]	编辑资源对象的属性，在线更新

续表

子命令	语 法	说 明
exec	kubectl exec POD [-c CONTAINER] -- COMMAND [args...] [flags]	执行一个容器中的命令
explain	kubectl explain RESOURCE [flags]	对资源对象属性的详细说明
expose	kubectl expose (-f FILENAME TYPE NAME) [--port=port] [--protocol=TCP UDP] [--target-port=number-or-name] [--name=name] [--external-ip=external-ip-of-service] [-type=type] [flags]	将已经存在的一个 RC、Service、Deployment 或 Pod 暴露为一个新的 Service
get	kubectl get [(-o --output=json yaml wide go-template=... go-template-file=... jsonpath=... jsonpath-file=...)] (TYPE [NAME -l label] TYPE/NAME ...) [flags]	显示一个或多个资源对象的概要信息
label	kubectl label [-ooverwrite] (-f FILENAME TYPE NAME) KEY_1=VAL_1 ... KEY_N=VAL_N [--resource-version=version] [flags]	设置或更新资源对象的 labels
logs	kubectl logs [-f] [-p] POD [-c CONTAINER] [flags]	屏幕打印一个容器的日志
namespace	kubectl namespace [namespace] [flags]	已被 kubectl config set-context 替代
patch	kubectl patch (-f FILENAME TYPE NAME) -p PATCH [flags]	以 merge 形式对资源对象的部分字段的值进行修改
port-forward	kubectl port-forward POD [LOCAL_PORT:]REMOTE_PORT [...[LOCAL_PORT_N:]REMOTE_PORT_N] [flags]	将本机的某个端口号映射到 Pod 的端口号，通常用于测试工作
proxy	kubectl proxy [--port=PORT] [--www=static-dir] [--www-prefix=prefix] [--api-prefix=prefix] [flags]	将本机某个端口号映射到 apiserver
replace	kubectl replace -f FILENAME [flags]	从配置文件或 stdin 替换资源对象
rolling-update	kubectl rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] --image=NEW_CONTAINER_IMAGE -f NEW_CONTROLLER_SPEC) [flags]	对 RC 进行滚动升级
rollout	kubectl rollout SUBCOMMAND [flags] kubectl rollout [command]	对 Deployment 进行管理，可用操作包括：history、pause、resume、undo、status
run	kubectl run NAME --image=image [--env="key=value"] [--port=port] [--replicas=replicas] [--dry-run=bool] [--overrides=inline-json] [--command] -- [COMMAND] [args...] [flags]	基于一个镜在 Kubernetes 集群上启动一个 Deployment
scale	kubectl scale [--resource-version=version] [--current-replicas=count] --replicas=COUNT (-f FILENAME TYPE NAME) [flags]	扩容、缩容一个 Deployment、ReplicaSet、RC 或 Job 中 Pod 的数量
set	kubectl set SUBCOMMAND [flags] kubectl set [command]	设置资源对象的某个特定信息，目前仅支持修改容器的镜像
taint	kubectl taint NODE NAME KEY_1=VAL_1:TINT_EFFECT_1 ... KEY_N=VAL_N:TINT_EFFECT_N [flags]	设置 Node 的 taint 信息，用于将特定的 Pod 调度到特定的 Node 的操作，为 Alpha 版本功能
uncordon	kubectl uncordon NODE [flags]	将 Node 设置为 schedulable
version	kubectl version [flags]	打印系统的版本信息

2.2.3 kubectl 参数列表

kubectl 命令行的公共启动参数如表 2.11 所示。

表 2.11 kubectl 命令行公共参数

参数名和取值示例	说 明
--alsologtostderr[=false]	设置为 true 表示将日志输出到文件的同时输出到 stderr
--as=""	设置本次操作的用户名
--certificate-authority=""	用于 CA 授权的 cert 文件路径
--client-certificate=""	用于 TLS 的客户端证书文件路径
--client-key=""	用于 TLS 的客户端 key 文件路径
--cluster=""	设置要使用的 kubeconfig 中的 cluster 名
--context=""	设置要使用的 kubeconfig 中的 context 名
--insecure-skip-tls-verify[=false]	设置为 true 表示跳过 TLS 安全验证模式，将使得 HTTPS 连接不安全
--kubeconfig=""	kubeconfig 配置文件路径，在配置文件中包括 Master 地址信息及必要的认证信息
--log-backtrace-at=:0	记录日志每到 “file:行号” 时打印一次 stack trace
--log-dir=""	日志文件路径
--log-flush-frequency=5s	设置 flush 日志文件的时间间隔
--logtostderr[=true]	设置为 true 表示将日志输出到 stderr，不输出到日志文件
--match-server-version[=false]	设置为 true 表示客户端版本号需要与服务端一致
--namespace=""	设置本次操作所在的 namespace
--password=""	设置 apiserver 的 basic authentication 的密码
-s, --server=""	设置 apiserver 的 URL 地址，默认为 localhost:8080
--stderrthreshold=2	在该 threshold 级别之上的日志将输出到 stderr
--token=""	设置访问 apiserver 的安全 token
--user=""	指定 kubeconfig 用户名
--username=""	设置 apiserver 的 basic authentication 的用户名
--v=0	glog 日志级别
--vmodule=	glog 基于模块的详细日志级别

每个子命令（如 create、delete、get 等）还有特定的 flags 参数，可以通过 \$ kubectl [command] --help 命令进行查看。

2.2.4 kubectl 输出格式

kubectl 命令可以用多种格式对结果进行显示，输出的格式通过 -o 参数指定：

```
$ kubectl [command] [TYPE] [NAME] -o=<output_format>
```

根据不同子命令的输出结果，可选的输出格式如表 2.12 所示。

表 2.12 kubectl 命令输出格式列表

输出格式	说 明
-o=custom-columns=<spec>	根据自定义列名进行输出，以逗号分隔
-o=custom-columns-file=<filename>	从文件中获取自定义列名进行输出
-o=json	以 JSON 格式显示结果
-o=jsonpath=<template>	输出 jsonpath 表达式定义的字段信息
-o=jsonpath-file=<filename>	输出 jsonpath 表达式定义的字段信息，来源于文件
-o=name	仅输出资源对象的名称
-o=wide	输出额外信息。对于 Pod，将输出 Pod 所在的 Node 名
-o=yaml	以 yaml 格式显示结果

常用的输出格式示例如下。

(1) 显示 Pod 的更多信息：

```
$ kubectl get pod <pod-name> -o wide
```

(2) 以 yaml 格式显示 Pod 的详细信息：

```
$ kubectl get pod <pod-name> -o yaml
```

(3) 以自定义列名显示 Pod 的信息：

```
$ kubectl get pod <pod-name>
-o=custom-columns=NAME:.metadata.name,RSRC:.metadata.resourceVersion
```

(4) 基于文件的自定义列名输出：

```
$ kubectl get pods <pod-name> -o=custom-columns-file=template.txt
```

template.txt 文件的内容为：

NAME	RSRC
metadata.name	metadata.resourceVersion

输出结果为：

NAME	RSRC
pod-name	52305

另外，还可以将输出结果按某个字段排序，通过--sort-by 参数以 jsonpath 表达式进行指定：

```
$ kubectl [command] [TYPE] [NAME] --sort-by=<jsonpath_exp>
```

例如，按照名字进行排序：

```
$ kubectl get pods --sort-by=.metadata.name
```

2.2.5 kubectl 操作示例

本节对一些常用的 kubectl 操作进行示例。

1. 创建资源对象

根据 yaml 配置文件一次性创建 service 和 rc:

```
$ kubectl create -f my-service.yaml -f my-rc.yaml
```

根据<directory>目录下所有.yaml、.yml、.json 文件的定义进行创建操作:

```
$ kubectl create -f <directory>
```

2. 查看资源对象

查看所有 Pod 列表:

```
$ kubectl get pods
```

查看 rc 和 service 列表:

```
$ kubectl get rc,service
```

3. 描述资源对象

显示 Node 的详细信息:

```
$ kubectl describe nodes <node-name>
```

显示 Pod 的详细信息:

```
$ kubectl describe pods/<pod-name>
```

显示由 RC 管理的 Pod 的信息:

```
$ kubectl describe pods <rc-name>
```

4. 删除资源对象

基于 pod.yaml 定义的名称删除 Pod:

```
$ kubectl delete -f pod.yaml
```

删除所有包含某个 label 的 Pod 和 service:

```
$ kubectl delete pods,services -l name=<label-name>
```

删除所有 Pod:

```
$ kubectl delete pods --all
```

5. 执行容器的命令

执行 Pod 的 date 命令， 默认使用 Pod 中的第 1 个容器执行：

```
$ kubectl exec <pod-name> date
```

指定 Pod 中某个容器执行 date 命令：

```
$ kubectl exec <pod-name> -c <container-name> date
```

通过 bash 获得 Pod 中某个容器的 TTY， 相当于登录容器：

```
$ kubectl exec -ti <pod-name> -c <container-name> /bin/bash
```

6. 查看容器的日志

查看容器输出到 stdout 的日志：

```
$ kubectl logs <pod-name>
```

跟踪查看容器的日志， 相当于 tail -f 命令的结果：

```
$ kubectl logs -f <pod-name> -c <container-name>
```

2.3 Guestbook 示例：Hello World

在对 Kubernetes 的容器应用进行详细说明之前，让我们先通过一个由 3 个微服务组成的留言板（Guestbook）系统的搭建，对 Kubernetes 对容器应用的基本操作和用法进行初步介绍。本章后面的章节将基于该案例和其他示例，进一步深入 Pod、RC、Service 等核心对象的用法和技巧，对 Kubernetes 的应用管理进行全面讲解。

Guestbook 留言板系统将通过 Pod、RC、Service 等资源对象搭建完成，成功启动后在网页中显示一条“Hello World”留言。其系统架构是一个基于 PHP+Redis 的分布式 Web 应用，前端 PHP Web 网站通过访问后端的 Redis 来完成用户留言的查询和添加等功能。同时 Redis 以 Master+Slave 的模式进行部署，实现数据的读写分离能力。

留言板系统的部署架构如图 2.4 所示。Web 层是一个基于 PHP 页面的 Apache 服务，启动 3 个实例组成集群，为客户端（例如浏览器）对网站的访问提供负载均衡。Redis Master 启动 1 个实例用于写操作（添加留言），Redis Slave 启动两个实例用于读操作（读取留言）。Redis Master 与 Slave 的数据同步由 Redis 具备的数据同步机制完成。

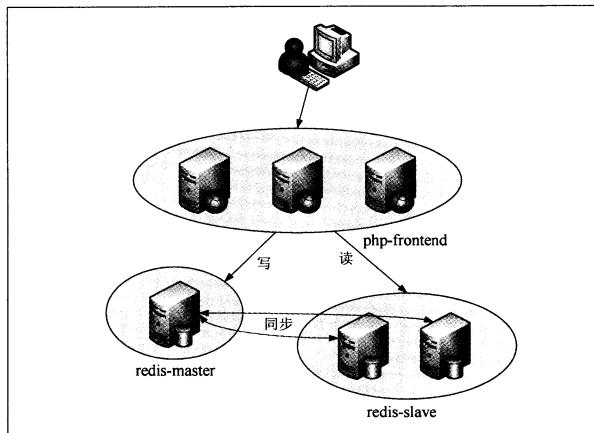


图 2.4 留言板的系统部署架构图

在本例中将要用到 3 个 Docker 镜像，下载地址为 <https://hub.docker.com/u/kubeguide/>。

- ◎ **redis-master**: 用于前端 Web 应用进行“写”留言操作的 Redis 服务，其中已经保存了一条内容为“Hello World!”的留言。
- ◎ **guestbook-redis-slave**: 用于前端 Web 应用进行“读”留言操作的 Redis 服务，并与 Redis-Master 的数据保持同步。
- ◎ **guestbook-php-frontend**: PHP Web 服务，在网页上展示留言的内容，也提供一个文本输入框供访客添加留言。

如图 2.5 所示为 Hello World 案例所采用的 Kubernetes 部署架构，这里 Master 与 Node 的服务处于同一个虚拟机中。通过创建 redis-master 服务、redis-slave 服务和 php-frontend 服务来实现整个系统的搭建。

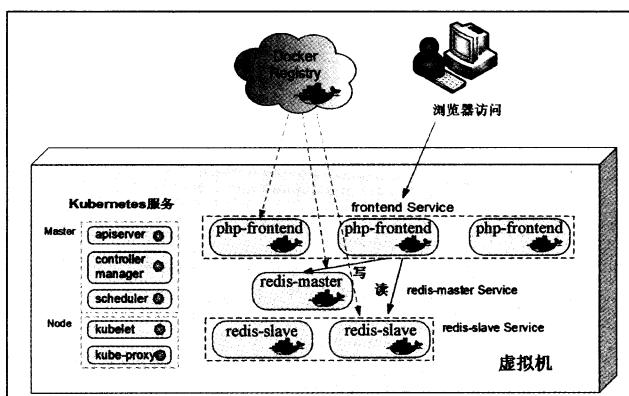


图 2.5 Kubernetes 部署架构图

2.3.1 创建 redis-master RC 和 Service

我们可以先定义 Service，然后定义一个 RC 来创建和控制相关联的 Pod，或者先定义 RC 来创建 Pod，然后定义与之关联的 Service，这里我们采用后一种方法。

首先为 redis-master 创建一个名为 redis-master 的 RC 定义文件 `redis-master-controller.yaml`。yaml 的语法类似于 PHP 的语法，对于空格的个数有严格的要求，详见 <http://yaml.org>。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master
  template:
    metadata:
      labels:
        name: redis-master
    spec:
      containers:
        - name: master
          image: kubeguide/redis-master
        ports:
          - containerPort: 6379
```

其中，`kind` 字段的值为“`ReplicationController`”，表示这是一个 RC；`spec.selector` 是 RC 的 Pod 选择器，即监控和管理拥有这些标签（Label）的 Pod 实例，确保当前集群上始终有且仅有 `replicas` 个 Pod 实例在运行，这里我们设置 `replicas=1` 表示只运行一个（名为 `redis-master` 的）Pod 实例，当集群中运行的 Pod 数量小于 `replicas` 时，RC 会根据 `spec.template` 段定义的 Pod 模板来生成一个新的 Pod 实例，`labels` 属性指定了该 Pod 的标签，注意，这里的 `labels` 必须匹配 RC 的 `spec.selector`，否则此 RC 就会陷入“只为他人做嫁衣”的悲惨世界中，永无翻身之时。

创建好 `redis-master-controller.yaml` 文件以后，我们在 Master 节点执行命令：`kubectl create -f <config_file>`，将它发布到 Kubernetes 集群中，就完成了 `redis-master` 的创建过程：

```
$ kubectl create -f redis-master-controller.yaml
replicationcontroller "redis-master" created
```

系统提示“`redis-master`”表示创建成功。然后我们用 `kubectl` 命令查看刚刚创建的 `redis-master`：

```
$ kubectl get rc
```

NAME	DESIRED	CURRENT	AGE
redis-master	1	1	5m

接下来运行 `kubectl get pods` 命令来查看当前系统中的 Pod 列表信息，我们看到一个名为 `redis-master-xxxxx` 的 Pod 实例，这是 Kubernetes 根据 `redis-master` 这个 RC 的定义自动创建的 Pod。RC 会给每个 Pod 实例在用户设置的 name 后补充一段 UUID，以区分不同的实例。由于 Pod 的调度和创建需要花费一定的时间，比如需要一定的时间来确定调度到哪个节点上，以及下载 Pod 的相关镜像，所以一开始我们看到 Pod 的状态将显示为 Pending。当 Pod 成功创建完成以后，状态会被更新为 Running。如果 Pod 一直处于 Pending 状态，则请参看第 5 章的查错说明。

```
$ kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
redis-master-b03io   1/1     Running   0          1h
```

`redis-master` Pod 已经创建并正常运行了，接下来我们就创建一个与之关联的 Service（服务）定义文件（文件名为 `redis-master-service.yaml`），完整的内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    name: redis-master
```

其中 `metadata.name` 是 Service 的服务名（ServiceName），`spec.selector` 确定了选择哪些 Pod，本例中的定义表明将选择设置过 `name=redis-master` 标签的 Pod。`port` 属性定义的是 Service 的虚拟端口号，`targetPort` 属性指定后端 Pod 内容器应用监听的端口号。

运行 `kubectl create` 命令创建该 service：

```
$ kubectl create -f redis-master-service.yaml
service "redis-master" created
```

系统提示“`service "redis-master" created`”表示创建成功。然后运行 `kubectl get` 命令可以查看到刚刚创建的 service：

```
$ kubectl get services
NAME            CLUSTER-IP      EXTERNAL-IP    PORT(S)        AGE
redis-master    169.169.208.57  <none>        6379/TCP     13m
```

注意到 redis-master 服务被分配了一个值为 169.169.208.57 的虚拟 IP 地址，随后，Kubernetes 集群中其他新创建的 Pod 就可以通过这个虚拟 IP 地址+端口 6379 来访问这个服务了。在本例中将要创建的 redis-slave 和 frontend 两组 Pod 都将通过 169.169.208.57:6379 来访问 redis-master 服务。

但由于 IP 地址是在服务创建后由 Kubernetes 系统自动分配的，在其他 Pod 中无法预先知道某个 Service 的虚拟 IP 地址，因此需要一个机制来找到这个服务。为此，Kubernetes 巧妙地使用了 Linux 环境变量（Environment Variable），在每个 Pod 的容器里都增加了一组 Service 相关的环境变量，用来记录从服务名到虚拟 IP 地址的映射关系。以 redis-master 服务为例，在容器的环境变量中会增加下面两条记录：

```
REDIS_MASTER_SERVICE_HOST=169.169.144.74  
REDIS_MASTER_SERVICE_PORT=6379
```

于是，redis-slave 和 frontend 等 Pod 中的应用程序就可以通过环境变量 REDIS_MASTER_SERVICE_HOST 得到 redis-master 服务的虚拟 IP 地址，通过环境变量 REDIS_MASTER_SERVICE_PORT 得到 redis-master 服务的端口号，这样就完成了对服务地址的查询功能。

2.3.2 创建 redis-slave RC 和 Service

现在我们已经成功启动了 redis-master 服务，接下来我们继续完成 redis-slave 服务的创建过程。在本案例中会启动 redis-slave 服务的两个副本，每个副本上的 Redis 进程都与 redis-master 进行数据同步，与 redis-master 共同组成了一个具备读写分离能力的 Redis 集群。留言板的 PHP 网页将通过访问 redis-slave 服务来读取留言数据。与之前的 redis-master 服务的创建过程一样，首先创建一个名为 redis-slave 的 RC 定义文件 redis-slave-controller.yaml，完整内容如下：

```
apiVersion: v1  
kind: ReplicationController  
metadata:  
  name: redis-slave  
  labels:  
    name: redis-slave  
spec:  
  replicas: 2  
  selector:  
    name: redis-slave  
  template:  
    metadata:  
      labels:  
        name: redis-slave  
    spec:  
      containers:
```

```

- name: slave
  image: kubeguide/guestbook-redis-slave
  env:
    - name: GET_HOSTS_FROM
      value: env
  ports:
    - containerPort: 6379

```

在容器的配置部分设置了一个环境变量 `GET_HOSTS_FROM=env`，意思是将从环境变量中获取 redis-master 服务的 IP 地址信息。

redis-slave 镜像中的启动脚本/run.sh 的内容为：

```

if [[ ${GET_HOSTS_FROM:-dns} == "env" ]]; then
  redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST} 6379
else
  redis-server --slaveof redis-master 6379
fi

```

在创建 redis-slave Pod 时，系统将自动在容器内部生成之前已经创建好的 redis-master service 相关的环境变量，所以 redis-slave 应用程序 redis-server 可以直接使用环境变量 `REDIS_MASTER_SERVICE_HOST` 来获取 redis-master 服务的 IP 地址。

如果在容器配置部分不设置该 env，则将使用 redis-master 服务的名称“redis-master”来访问它，这将使用 DNS 方式的服务发现，需要预先启动 Kubernetes 集群的 skydns 服务，详见 2.5.4 节的说明。

运行 `kubectl create` 命令：

```
$ kubectl create -f redis-slave-controller.yaml
ReplicationControllers "redis-slave" created
```

运行 `kubectl get` 命令查看 RC：

```
$ kubectl get rc
NAME          DESIRED   CURRENT   AGE
redis-master   1         1         1h
redis-slave    2         2         1h
```

查看 RC 创建的 Pod，可以看到有两个 redis-slave Pod 在运行：

```
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
redis-master-b03io  1/1    Running   0          1h
redis-slave-10ahl  1/1    Running   0          1h
redis-slave-c5y10  1/1    Running   0          1h
```

然后创建 redis-slave 服务。类似于 redis-master 服务，与 redis-slave 相关的一组环境变量也将再后续新建的 frontend Pod 中由系统自动生成。

配置文件 `redis-slave-service.yaml` 的内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  ports:
  - port: 6379
  selector:
    name: redis-slave
```

运行 `kubectl` 创建 Service：

```
$ kubectl create -f redis-slave-service.yaml
services/redis-slave
```

通过 `kubectl` 查看创建的 Service：

```
$ kubectl get services
NAME           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
frontend       169.169.167.153 <nodes>        80/TCP        25m
redis-master   169.169.208.57  <none>         6379/TCP     25m
redis-slave    169.169.78.102 <none>         6379/TCP     25m
```

2.3.3 创建 frontend RC 和 Service

类似地，定义 `frontend` 的 RC 配置文件——`frontend-controller.yaml`，内容如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
      - name: frontend
```

```

image: kubeguide/guestbook-php-frontend
env:
- name: GET_HOSTS_FROM
  value: env
ports:
- containerPort: 80

```

在容器的配置部分设置了一个环境变量 `GET_HOSTS_FROM=env`，意思是如果从环境变量中获取 `redis-master` 和 `redis-slave` 服务的 IP 地址信息。

容器镜像名为 `kubeguide/guestbook-php-frontend`，该镜像中所包含的 PHP 的留言板源码 (`guestbook.php`) 如下：

```

<?
set_include_path('.:./usr/local/lib/php');
error_reporting(E_ALL);
ini_set('display_errors', 1);
require 'Predis/Autoloader.php';
Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
    $host = 'redis-master';
    if (getenv('GET_HOSTS_FROM') == 'env') {
        $host = getenv('REDIS_MASTER_SERVICE_HOST');
    }
    header('Content-Type: application/json');
    if ($_GET['cmd'] == 'set') {
        $client = new Predis\Client([
            'scheme' => 'tcp',
            'host'   => $host,
            'port'   => 6379,
        ]);

        $client->set($_GET['key'], $_GET['value']);
        print('{"message": "Updated"}');
    } else {
        $host = 'redis-slave';
        if (getenv('GET_HOSTS_FROM') == 'env') {
            $host = getenv('REDIS_SLAVE_SERVICE_HOST');
        }
        $client = new Predis\Client([
            'scheme' => 'tcp',
            'host'   => $host,
            'port'   => 6379,
        ]);

        $value = $client->get($_GET['key']);
    }
}

```

```

    print('{"data": "' . $value . '"}')) ;
}
} else {
    phpinfo();
} ?>

```

这段 PHP 代码的意思是，如果是一个 set 请求（提交留言），则会连接到 redis-master 服务进行写数据操作，其中 redis-master 服务的虚拟 IP 地址是用之前提过的从环境变量中获取的方式得到的，端口使用默认的 6379 端口号（当然，也可以使用环境变量'REDIS_MASTER_SERVICE_PORT'的值）；如果是 get 请求，则会连接到 redis-slave 服务进行读数据操作。

可以看到，如果在容器配置部分不设置 env “GET_HOSTS_FROM”，则将使用 redis-master 或 redis-slave 服务名来访问这两个服务，这将使用 DNS 方式的服务发现，需要预先启动 Kubernetes 集群的 skydns 服务，详见 2.5.4 节的说明。

运行 kubectl create 命令创建 RC:

```
$ kubectl create -f frontend-controller.yaml
replicationcontrollers "frontend" created
```

查看已创建的 RC:

```
$ kubectl get rc
NAME      DESIRED   CURRENT   AGE
frontend   3         3         1h
redis-master 1         1         1h
redis-slave  2         2         1h
```

再查看生成的 Pod:

```
$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
redis-master-b03io  1/1     Running   0          1h
redis-slave-10ahl  1/1     Running   0          1h
redis-slave-c5y10  1/1     Running   0          1h
frontend-4o11g    1/1     Running   0          1h
frontend-u9aq6    1/1     Running   0          1h
frontend-ygall    1/1     Running   0          1h
```

最后创建 frontend Service，主要目的是使用 Service 的 NodePort 给 Kubernetes 集群中的 Service 映射一个外网可以访问的端口，这样一来，外部网络就可以通过 NodeIP+NodePort 的方式访问集群中的服务了。

服务定义文件 frontend-service.yaml 的内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
```

```

labels:
  name: frontend
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30001
  selector:
    name: frontend

```

这里的关键点是设置 `type=NodePort` 并指定一个 `NodePort` 的值，表示使用 Node 上的物理机端口提供对外访问的能力。需要注意的是，`spec.ports.NodePort` 的端口号范围可以进行限制（通过 `kube-apiserver` 的启动参数`--service-node-port-range` 指定），默认为 30000~32767，如果指定为可用 IP 范围之外的其他端口号，则 Service 的创建将会失败。

运行 `kubectl create` 命令创建 Service：

```
$ kubectl create -f frontend-service.yaml
Services "frontend" created
```

通过 `kubectl` 查看创建的 Service：

```
$ kubectl get services
NAME           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
frontend       169.169.167.153 <nodes>        80/TCP        25m
redis-master   169.169.208.57  <none>         6379/TCP     25m
redis-slave    169.169.78.102 <none>         6379/TCP     25m
```

2.3.4 通过浏览器访问 frontend 页面

经过上面的三个步骤就搭建好了 Guestbook 留言板系统，总共包括 3 个应用的 6 个实例，都运行在 Kubernetes 集群中。打开浏览器，在地址栏输入 `http://虚拟机 IP:30001/`，将看到如图 2.6 所示的网页，并且看到网页上有一条留言——“Hello World!”。

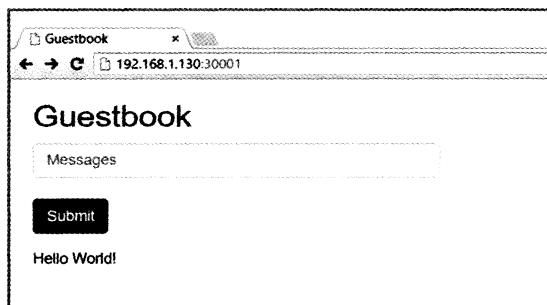


图 2.6 通过浏览器访问留言板网页

尝试输入一条新的留言“Hi Kubernetes!”，单击 Submit 按钮，网页将会在原留言的下方显示新的留言，说明这条留言已经被成功加入 Redis 数据库中了，如图 2.7 所示。

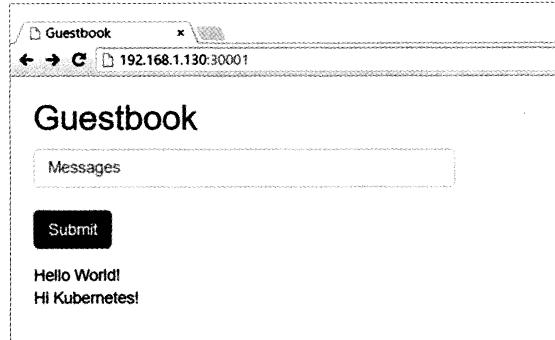


图 2.7 在留言板网页添加新的留言

通过 Guestbook 示例，可以看到 Kubernetes 强大的应用管理功能，用户仅需通过几个简单的 YAML 配置就能完成复杂系统的搭建，并能够通过 Kubernetes 自动实现服务发现和负载均衡。接下来，让我们深入 Pod 的应用、配置、调度管理及服务的应用，开始 Kubernetes 应用管理之旅。

2.4 深入掌握 Pod

本节将对 Kubernetes 如何发布和管理应用进行详细说明和示例，主要包括 Pod 和容器的使用、Pod 的控制和调度管理、应用配置管理等内容。

2.4.1 Pod 定义详解

yaml 格式的 Pod 定义文件的完整内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: string
  namespace: string
  labels:
    - name: string
  annotations:
    - name: string
spec:
```

```
containers:
- name: string
  image: string
  imagePullPolicy: [Always | Never | IfNotPresent]
  command: [string]
  args: [string]
  workingDir: string
  volumeMounts:
- name: string
  mountPath: string
  readOnly: boolean
  ports:
- name: string
  containerPort: int
  hostPort: int
  protocol: string
  env:
- name: string
  value: string
  resources:
    limits:
      cpu: string
      memory: string
    requests:
      cpu: string
      memory: string
  livenessProbe:
    exec:
      command: [string]
    httpGet:
      path: string
      port: number
      host: string
      scheme: string
      httpHeaders:
- name: string
  value: string
    tcpSocket:
      port: number
    initialDelaySeconds: 0
    timeoutSeconds: 0
    periodSeconds: 0
    successThreshold: 0
    failureThreshold: 0
```

```

securityContext:
  privileged: false
restartPolicy: [Always | Never | OnFailure]
nodeSelector: object
imagePullSecrets:
- name: string
hostNetwork: false
volumes:
- name: string
  emptyDir: {}
  hostPath:
    path: string
  secret:
    secretName: string
    items:
    - key: string
      path: string
  configMap:
    name: string
    items:
    - key: string
      path: string

```

对各属性的详细说明如表 2.13 所示。

表 2.13 对 Pod 定义文件模板中各属性的详细说明

属性名称	取值类型	是否必选	取值说明
version	String	Required	版本号, 例如 v1
kind	String	Required	Pod
metadata	Object	Required	元数据
metadata.name	String	Required	Pod 的名称, 命名规范需符合 RFC 1035 规范
metadata.namespace	String	Required	Pod 所属的命名空间, 默认为 “default”
metadata.labels[]	List		自定义标签列表
metadata.annotation[]	List		自定义注解列表
Spec	Object	Required	Pod 中容器的详细定义
spec.containers[]	List	Required	Pod 中的容器列表
spec.containers[].name	String	Required	容器的名称, 需符合 RFC 1035 规范
spec.containers[].image	String	Required	容器的镜像名称

续表

属性名称	取值类型	是否必选	取值说明
spec.containers[].imagePullPolicy	String		获取镜像的策略，可选值包括：Always、Never、IfNotPresent，默认值为 Always。 Always：表示每次都尝试重新下载镜像。 IfNotPresent：表示如果本地有该镜像，则使用本地的镜像，本地不存在时下载镜像。 Never：表示仅使用本地镜像
spec.containers[].command[]	List		容器的启动命令列表，如果不指定，则使用镜像打包时使用的启动命令
spec.containers[].args[]	List		容器的启动命令参数列表
spec.containers[].workingDir	String		容器的工作目录
spec.containers[].volumeMounts[]	List		挂载到容器内部的存储卷配置
spec.containers[].volumeMounts[].name	String		引用 Pod 定义的共享存储卷的名称，需使用 volumes[] 部分定义的共享存储卷名称
spec.containers[].volumeMounts[].mountPath	String		存储卷在容器内 Mount 的绝对路径，应少于 512 个字符
spec.containers[].volumeMounts[].readOnly	Boolean		是否为只读模式，默认为读写模式
spec.containers[].ports[]	List		容器需要暴露的端口号列表
spec.containers[].ports[].name	String		端口的名称
spec.containers[].ports[].containerPort	Int		容器需要监听的端口号
spec.containers[].ports[].hostPort	Int		容器所在主机需要监听的端口号，默认与 containerPort 相同。设置 hostPort 时，同一台宿主机将无法启动该容器的第 2 份副本
spec.containers[].ports[].protocol	String		端口协议，支持 TCP 和 UDP，默认为 TCP
spec.containers[].env[]	List		容器运行前需设置的环境变量列表
spec.containers[].env[].name	String		环境变量的名称
spec.containers[].env[].value	String		环境变量的值
spec.containers[].resources	Object		资源限制和资源请求的设置，详见第 5 章的说明
spec.containers[].resources.limits	Object		资源限制的设置
spec.containers[].resources.limits.cpu	String		CPU 限制，单位为 core 数，将用于 docker run --cpu-shares 参数
spec.containers[].resources.limits.memory	String		内存限制，单位可以为 MiB/GiB 等，将用于 docker run --memory 参数
spec.containers[].resources.requests	Object		资源限制的设置

续表

属性名称	取值类型	是否必选	取值说明
spec.containers[].resources.requests.cpu	String		CPU 请求，单位为 core 数，容器启动的初始可用数量
spec.containers[].resources.requests.memory	String		内存请求，单位可以为 MiB、GiB 等，容器启动的初始可用数量
spec.volumes[]	List		在该 Pod 上定义的共享存储卷列表
spec.volumes[].name	String		共享存储卷的名称，在一个 Pod 中每个存储卷定义一个名称，应符合 RFC 1035 规范。容器定义部分的 containers[].volumeMounts[].name 将引用该共享存储卷的名称。 volume 的类型包括：emptyDir、hostPath、gcePersistentDisk、awsElasticBlockStore、gitRepo、secret、nfs、iscsi、glusterfs、persistentVolumeClaim、rbd、flexVolume、cinder、cephfs、flocker、downwardAPI、fc、azureFile、configMap、vsphereVolume，可以定义多个 volume，每个 volume 的 name 保持唯一。本节讲解 emptyDir、hostPath、secret、configMap 这 4 种 volume，其他类型 volume 的设置方式详见第 1 章的说明
spec.volumes[].emptyDir	Object		类型为 emptyDir 的存储卷，表示与 Pod 同生命周期的一个临时目录，其值为一个空对象：emptyDir: {}
spec.volumes[].hostPath	Object		类型为 hostPath 的存储卷，表示挂载 Pod 所在宿主机的目录，通过 volumes[].hostPath.path 指定
spec.volumes[].hostPath.path	String		Pod 所在主机的目录，将被用于容器中 mount 的目录
spec.volumes[].secret	Object		类型为 secret 的存储卷，表示挂载集群预定义的 secret 对象到容器内部
spec.volumes[].configMap	Object		类型为 configMap 的存储卷，表示挂载集群预定的 configMap 对象到容器内部
spec.volumes[].livenessProbe	Object		对 Pod 内各容器健康检查的设置，当探测无响应几次之后，系统将自动重启该容器。可以设置的方法包括：exec、httpGet 和 tcpSocket。对一个容器仅需设置一种健康检查方法
spec.volumes[].livenessProbe.exec	Object		对 Pod 内各容器健康检查的设置，exec 方式
spec.volumes[].livenessProbe.exec.command[]	String		exec 方式需要指定的命令或者脚本
spec.volumes[].livenessProbe.httpGet	Object		对 Pod 内各容器健康检查的设置，HTTPGet 方式。需指定 path、port

续表

属性名称	取值类型	是否必选	取值说明
spec.volumes[].livenessProbe.tcpSocket	Object		对 Pod 内各容器健康检查的设置，tcpSocket 方式
spec.volumes[].livenessProbe.initialDelaySeconds	Number		容器启动完成后进行首次探测的时间，单位为秒
spec.volumes[].livenessProbe.timeoutSeconds	Number		对容器健康检查的探测等待响应的超时时间设置，单位为秒，默认为 1 秒。超过该超时时间设置，将认为该容器不健康，将重启该容器
spec.volumes[].livenessProbe.periodSeconds	Number		对容器健康检查的定期探测时间设置，单位为秒，默认为 10 秒探测一次
spec.restartPolicy	String		Pod 的重启策略，可选值为 Always、OnFailure， 默认值为 Always。 Always：Pod 一旦终止运行，则无论容器是如何终止的，kubelet 都将重启它。 OnFailure：只有 Pod 以非零退出码终止时，kubelet 才会重启该容器。如果容器正常结束（退出码为 0），则 kubelet 将不会重启它。 Never：Pod 终止后，kubelet 将退出码报告给 Master，不会再重启该 Pod
spec.nodeSelector	Object		设置 NodeSelector 表示将该 Pod 调度到包含这些 label 的 Node 上，以 key:value 格式指定
spec.imagePullSecrets	Object		Pull 镜像时使用的 secret 名称，以 name:secretkey 格式指定
spec.hostNetwork	Boolean		是否使用主机网络模式，默认为 false。如果设置为 true，则表示容器使用宿主机网络，不再使用 Docker 网桥，该 Pod 将无法在同一台宿主机上启动第 2 个副本

2.4.2 Pod 的基本用法

在对 Pod 的用法进行说明之前，有必要先对 Docker 容器中应用的运行要求进行说明。

在使用 Docker 时，可以使用 docker run 命令创建并启动一个容器。而在 Kubernetes 系统中对长时间运行容器的要求是：其主程序需要一直在前台执行。如果我们创建的 Docker 镜像的启动命令是后台执行程序，例如 Linux 脚本：

```
nohup ./start.sh &
```

则在 kubelet 创建包含这个容器的 Pod 之后运行完该命令，即认为 Pod 执行结束，将立刻销毁该 Pod。如果为该 Pod 定义了 ReplicationController，则系统将会监控到该 Pod 已经终止，之后根

据 RC 定义中 Pod 的 replicas 副本数量生成一个新的 Pod。而一旦创建出新的 Pod，就将在执行完启动命令后，陷入无限循环的过程中。这就是 Kubernetes 需要我们自己创建的 Docker 镜像以一个前台命令作为启动命令的原因。

对于无法改造为前台执行的应用，也可以使用开源工具 Supervisor 辅助进行前台运行的功能。Supervisor 提供了一种可以同时启动多个后台应用，并保持 Supervisor 自身在前台执行的机制，可以满足 Kubernetes 对容器的启动要求。关于 Supervisor 的安装和使用，请参考官网 <http://supervisord.org> 的文档说明。

接下来对 Pod 对容器的封装和应用进行说明，Pod 的基本用法为：Pod 可以由 1 个或多个容器组合而成。

在上一节 Guestbook 的例子中，名为 frontend 的 Pod 只由一个容器组成：

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  containers:
  - name: frontend
    image: kubeguide/guestbook-php-frontend
    env:
    - name: GET_HOSTS_FROM
      value: env
    ports:
    - containerPort: 80
```

最新网络工程师资料
www.wlgcs.cn

这个 frontend Pod 在成功启动之后，将启动 1 个 Docker 容器。

另一种场景是，当 frontend 和 redis 两个容器应用为紧耦合的关系，应该组合成一个整体对外提供服务时，则应将这两个容器打包为一个 Pod，如图 2.8 所示。

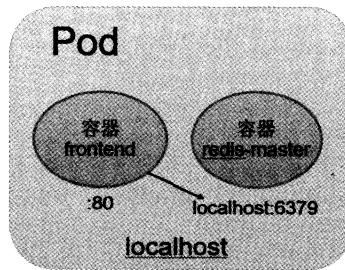


图 2.8 包含两个容器的 Pod

配置文件 `frontend-localredis-pod.yaml` 如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-php
  labels:
    name: redis-php
spec:
  containers:
    - name: frontend
      image: kubeguide/guestbook-php-frontend:localredis
      ports:
        - containerPort: 80
    - name: redis
      image: kubeguide/redis-master
      ports:
        - containerPort: 6379
```

属于一个 Pod 的多个容器应用之间相互访问时仅需要通过 `localhost` 就可以通信，使得这一组容器被“绑定”在了一个环境中。

在 Docker 容器 `kubeguide/guestbook-php-frontend:localredis` 的 PHP 网页中，直接通过 URL 地址“`localhost:6379`”对同属于一个 Pod 内的 `redis-master` 进行访问。`guestbook.php` 的内容如下：

```
<?
set_include_path('.:./usr/local/lib/php');
error_reporting(E_ALL);
ini_set('display_errors', 1);
require 'Predis/Autoloader.php';
Predis\Autoloader::register();

if (isset($_GET['cmd']) === true) {
    $host = 'localhost';
    if (getenv('REDIS_HOST') && strlen(getenv('REDIS_HOST')) > 0 ) {
        $host = getenv('REDIS_HOST');
    }
    header('Content-Type: application/json');
    if ($_GET['cmd'] == 'set') {
        $client = new Predis\Client([
            'scheme' => 'tcp',
            'host'   => $host,
            'port'   => 6379,
        ]);
        $client->set($_GET['key'], $_GET['value']);
        print('{"message": "Updated"}');
    } else {
}
```

```

$host = 'localhost';
if (getenv('REDIS_HOST') && strlen(getenv('REDIS_HOST')) > 0 ) {
    $host = getenv('REDIS_HOST');
}
$client = new Predis\Client([
    'scheme' => 'tcp',
    'host'   => $host,
    'port'   => 6379,
]);
$value = $client->get($_GET['key']);
print('{"data": "' . $value . '"}');
}
} else {
    phpinfo();
} ?>

```

运行 `kubectl create` 命令创建该 Pod:

```
$ kubectl create -f frontend-localredis-pod.yaml
pod "redis-php" created
```

查看已创建的 Pod:

```
# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
redis-php  2/2      Running   0          10m
```

可以看到 READY 信息为 2/2，表示 Pod 中的两个容器都成功运行了。

查看这个 Pod 的详细信息，可以看到两个容器的定义及创建的过程（Event 事件信息）：

```
# kubectl describe pod redis-php
Name:           redis-php
Namespace:      default
Node:          k8s/192.168.18.3
Start Time:    Thu, 28 Jul 2016 12:28:21 +0800
Labels:        name=redis-php
Status:        Running
IP:            172.17.1.4
Controllers:   <none>
Containers:
  frontend:
    Container ID: docker://ccc8616f8df1fb19abbd0ab189a36e6f6628b78ba7b97b1077d86e7fc224ee08
    Image:          kubeguide/guestbook-php-frontend:localredis
    Image ID:      docker://sha256:d014f67384a11186e135b95a7ed0d794674f7ce258f0dce47267c3052a0d0fa9
    Port:          80/TCP
    State:         Running
```

```

Started:           Thu, 28 Jul 2016 12:28:22 +0800
Ready:            True
Restart Count:    0
Environment Variables: <none>
redis:
  Container ID: docker://c0b19362097cda6dd5b8ed7d8eaaaf43aeeb969ee023ef255604bde089808075
    Image:          kubeguide/redis-master
    Image ID:       docker://sha256:405a0b586f7eb545ec65be0e914311159d1baedccd3a93e9d3e3b249ec5cbd
      Port:          6379/TCP
      State:         Running
      Started:       Thu, 28 Jul 2016 12:28:23 +0800
      Ready:          True
      Restart Count: 0
      Environment Variables: <none>
    Conditions:
      Type     Status
      Initialized  True
      Ready      True
      PodScheduled  True
    Volumes:
      default-token-97j21:
        Type:      Secret (a volume populated by a Secret)
        SecretName: default-token-97j21
      QoS Tier:  BestEffort
    Events:
      FirstSeen  LastSeen  Count  From      SubobjectPath  Type  Reason  Message
      -----  -----  -----  ----      -----  ----  -----  -----
      18m        18m      1  {default-scheduler }  Normal
      Scheduled  Successfully assigned redis-php to k8s-node-1
      18m        18m      1  {kubelet k8s-node-1}
      spec.containers{frontend}  Normal      Pulled          Container image
      "kubeguide/guestbook-php-frontend:localredis" already present on machine
      18m        18m      1  {kubelet k8s-node-1}
      spec.containers{frontend}  Normal      Created          Created container
      with docker id ccc8616f8df1
      18m        18m      1  {kubelet k8s-node-1}
      spec.containers{frontend}  Normal      Started          Started container
      with docker id ccc8616f8df1
      18m        18m      1  {kubelet k8s-node-1}
      spec.containers{redis}    Normal      Pulled          Container image
      "kubeguide/redis-master" already present on machine
      18m        18m      1  {kubelet k8s-node-1}
      spec.containers{redis}    Normal      Created          Created container
      with docker id c0b19362097c
      18m        18m      1  {kubelet k8s-node-1}

```

```
spec.containers{redis}           Normal        Started          Started container
with docker id c0b19362097c
```

2.4.3 静态 Pod

静态 Pod 是由 kubelet 进行管理的仅存在于特定 Node 上的 Pod。它们不能通过 API Server 进行管理，无法与 ReplicationController、Deployment 或者 DaemonSet 进行关联，并且 kubelet 也无法对它们进行健康检查。静态 Pod 总是由 kubelet 进行创建，并且总是在 kubelet 所在的 Node 上运行。

创建静态 Pod 有两种方式：配置文件或者 HTTP 方式。

1) 配置文件方式

首先，需要设置 kubelet 的启动参数 “--config”，指定 kubelet 需要监控的配置文件所在的目录，kubelet 会定期扫描该目录，并根据该目录中的.yaml 或.json 文件进行创建操作。

假设配置目录为 /etc/kubelet.d/，配置启动参数：--config=/etc/kubelet.d/，然后重启 kubelet 服务。

在目录 /etc/kubelet.d 中放入 static-web.yaml 文件，内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    name: static-web
spec:
  containers:
  - name: static-web
    image: nginx
    ports:
    - name: web
      containerPort: 80
```

等待一会儿，查看本机中已经启动的容器：

```
# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED      STATUS      PORTS      NAMES
2292ea231ab1   nginx     "nginx -g 'daemon off'"  1 minute ago   1m
k8s_static-web.68ee0075_static-web-k8s-node-1_default_78c7efddeb191c949cbb7aa22
a927c8_401b96d0
```

可以看到一个 Nginx 容器已经被 kubelet 成功创建了出来。

到 Master 节点查看 Pod 列表，可以看到这个 static pod：

```
# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
static-web-node1 1/1     Running   0          5m
```

由于静态 Pod 无法通过 API Server 直接管理，所以在 Master 节点尝试删除这个 Pod，将会使其变成 Pending 状态，且不会被删除。

```
# kubectl delete pod static-web-node1
pod "static-web-node1" deleted
```

```
# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
static-web-node1 0/1     Pending   0          1s
```

删除该 Pod 的操作只能是到其所在 Node 上，将其定义文件 static-web.yaml 从 /etc/kubelet.d 目录下删除。

```
# rm /etc/kubelet.d/static-web.yaml
# docker ps
// 无容器正在运行。
```

2.4.4 Pod 容器共享 Volume

在同一个 Pod 中的多个容器能够共享 Pod 级别的存储卷 Volume。Volume 可以定义为各种类型，多个容器各自进行挂载操作，将一个 Volume 挂载为容器内部需要的目录，如图 2.9 所示。

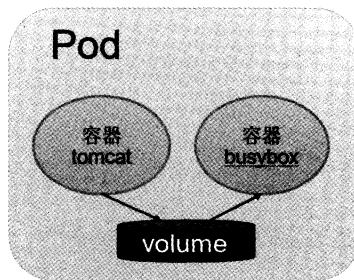


图 2.9 Pod 中多个容器共享 volume

在下面的例子中，Pod 内包含两个容器：tomcat 和 busybox，在 Pod 级别设置 Volume “app-logs”，用于 tomcat 向其中写日志文件，busybox 读日志文件。

配置文件 pod-volume-applogs.yaml 的内容如下：

```
apiVersion: v1
kind: Pod
metadata:
```

```

name: volume-pod
spec:
  containers:
    - name: tomcat
      image: tomcat
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: app-logs
          mountPath: /usr/local/tomcat/logs
    - name: busybox
      image: busybox
      command: ["sh", "-c", "tail -f /logs/catalina*.log"]
      volumeMounts:
        - name: app-logs
          mountPath: /logs
  volumes:
    - name: app-logs
      emptyDir: {}

```

这里设置的 Volume 名为 app-logs，类型为 emptyDir（也可以设置为其他类型，详见第1章对 Volume 概念的说明），挂载到 tomcat 容器内的 /usr/local/tomcat/logs 目录，同时挂载到 logreader 容器内的 /logs 目录。tomcat 容器在启动后会向 /usr/local/tomcat/logs 目录中写文件，logreader 容器就可以读取其中的文件了。

logreader 容器的启动命令为 tail -f /logs/catalina*.log，我们可以通过 kubectl logs 命令查看 logreader 容器的输出内容：

```
# kubectl logs volume-pod -c busybox
.....
29-Jul-2016 12:55:59.626 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application
directory /usr/local/tomcat/webapps/manager
29-Jul-2016 12:55:59.722 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web
application directory /usr/local/tomcat/webapps/manager has finished in 96 ms
29-Jul-2016 12:55:59.740 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["http-apr-8080"]
29-Jul-2016 12:55:59.794 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["ajp-apr-8009"]
29-Jul-2016 12:56:00.604 INFO [main] org.apache.catalina.startup.Catalina.start
Server startup in 4052 ms
```

这个文件即为 tomcat 生成的日志文件 /usr/local/tomcat/logs/catalina.<date>.log 的内容。登录 tomcat 容器进行查看：

```
# kubectl exec -ti volume-pod -c tomcat -- ls /usr/local/tomcat/logs
```

```
catalina.2016-07-29.log      localhost_access_log.2016-07-29.txt
host-manager.2016-07-29.log  manager.2016-07-29.log

# kubectl exec -ti volume-pod -c tomcat -- tail
/usr/local/tomcat/logs/catalina.2016-07-29.log
.....
29-Jul-2016 12:55:59.722 INFO [localhost-startStop-1]
org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web
application directory /usr/local/tomcat/webapps/manager has finished in 96 ms
29-Jul-2016 12:55:59.740 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["http-apr-8080"]
29-Jul-2016 12:55:59.794 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["ajp-apr-8009"]
29-Jul-2016 12:56:00.604 INFO [main] org.apache.catalina.startup.Catalina.start
Server startup in 4052 ms
```

2.4.5 Pod 的配置管理

应用部署的一个最佳实践是将应用所需的配置信息与程序进行分离，这样可以使得应用程序被更好地复用，通过不同的配置也能实现更灵活的功能。将应用打包为容器镜像后，可以通过环境变量或者外挂文件的方式在创建容器时进行配置注入，但在大规模容器集群的环境中，对多个容器进行不同的配置将变得非常复杂。Kubernetes v1.2 版本提供了一种统一的集群配置管理方案——ConfigMap。本节对 ConfigMap 的概念和用法进行详细描述。

1. ConfigMap：容器应用的配置管理

ConfigMap 供容器使用的典型用法如下。

- (1) 生成为容器内的环境变量。
- (2) 设置容器启动命令的启动参数（需设置为环境变量）。
- (3) 以 Volume 的形式挂载为容器内部的文件或目录。

ConfigMap 以一个或多个 key:value 的形式保存在 Kubernetes 系统中供应用使用，既可以用于表示一个变量的值（例如 appLogLevel=info），也可以用于表示一个完整配置文件的内容（例如 server.xml=<?xml...>...）

可以通过 yaml 配置文件或者直接使用 kubectl create configmap 命令行的方式来创建 ConfigMap。

2. ConfigMap的创建：yaml文件方式

下面的例子 cm-appvars.yaml 描述了将几个应用所需的变量定义为 ConfigMap 的用法：

```
cm-appvars.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appvars
data:
  apploglevel: info
  appdatadir: /var/data
```

执行 kubectl create 命令创建该 ConfigMap：

```
$ kubectl create -f cm-appvars.yaml
configmap "cm-appvars" created
```

查看创建好的 ConfigMap：

```
# kubectl get configmap
NAME      DATA   AGE
cm-appvars    2     3s

# kubectl describe configmap cm-appvars
Name:            cm-appvars
Namespace:       default
Labels:          <none>
Annotations:    <none>

Data
=====
appdatadir:    9 bytes
apploglevel:   4 bytes

# kubectl get configmap cm-appvars -o yaml
apiVersion: v1
data:
  appdatadir: /var/data
  apploglevel: info
kind: ConfigMap
metadata:
  creationTimestamp: 2016-07-28T19:57:16Z
  name: cm-appvars
  namespace: default
  resourceVersion: "78709"
  selfLink: /api/v1/namespaces/default/configmaps/cm-appvars
  uid: 7bb2e9c0-54fd-11e6-9dcd-000c29dc2102
```

下面的例子 `cm-appconfigfiles.yaml` 描述了将两个配置文件 `server.xml` 和 `logging.properties` 定义为 ConfigMap 的用法，设置 key 为配置文件的别名，value 则是配置文件的全部文本内容：

```
cm-appconfigfiles.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appconfigfiles
data:
  key-serverxml: |
    <?xml version='1.0' encoding='utf-8'?>
    <Server port="8005" shutdown="SHUTDOWN">
      <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
      <Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
      <Listener className=
"org.apache.catalina.core.JreMemoryLeakPreventionListener" />
      <Listener className=
"org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
      <Listener className=
"org.apache.catalina.core.ThreadLocalLeakPreventionListener" />
      <GlobalNamingResources>
        <Resource name="UserDatabase" auth="Container"
          type="org.apache.catalina.UserDatabase"
          description="User database that can be updated and saved"
          factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
          pathname="conf/tomcat-users.xml" />
      </GlobalNamingResources>

      <Service name="Catalina">
        <Connector port="8080" protocol="HTTP/1.1"
          connectionTimeout="20000"
          redirectPort="8443" />
        <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
        <Engine name="Catalina" defaultHost="localhost">
          <Realm className="org.apache.catalina.realm.LockOutRealm">
            <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
              resourceName="UserDatabase"/>
          </Realm>
          <Host name="localhost" appBase="webapps"
            unpackWARs="true" autoDeploy="true">
            <Valve className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
              prefix="localhost_access_log" suffix=".txt"
              pattern="%h %l %u %t &quot;%r&quot; %s %b" />
          </Host>
        </Engine>
      </Service>
    </Server>
  
```

```

        </Engine>
    </Service>
</Server>
key-loggingproperties: "handlers
    =lcatalina.org.apache.juli.FileHandler, 2localhost.org.apache.juli.
FileHandler,
    3manager.org.apache.juli.FileHandler, 4host-manager.org.apache.juli.
FileHandler,
    java.util.logging.ConsoleHandler\r\n\r\n.n.handlers= 1catalina.org.apache.
juli.FileHandler,
    java.util.logging.ConsoleHandler\r\n\r\n1catalina.org.apache.juli.FileHandler.level
    = FINE\r\n1catalina.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHandler.prefix
    = catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.level =
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
    = ${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHandler.prefix =
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.level
    = FINE\r\n3manager.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandler.prefix
    = manager.\r\n\r\n4host-manager.org.apache.juli.FileHandler.level =
FINE\r\n4host-manager.org.apache.juli.FileHandler.directory
    = ${catalina.base}/logs\r\n4host-manager.org.apache.juli.FileHandler.
prefix =
        host-manager.\r\n\r\njava.util.logging.ConsoleHandler.level = FINE\r\n\r\n
java.util.logging.ConsoleHandler.formatter
    = java.util.logging.SimpleFormatter\r\n\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].level
    = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
handlers
    = 2localhost.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/manager].level
    = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager].handlers
    = 3manager.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/host-manager].level
    = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/host-manager].handlers
    = 4host-manager.org.apache.juli.FileHandler\r\n\r\n"

```

执行 kubectl create 命令创建该 ConfigMap:

```
# kubectl create -f cm-appconfigfiles.yaml
configmap "cm-appconfigfiles" created
```

查看创建好的 ConfigMap:

```
# kubectl get configmap cm-appconfigfiles
NAME          DATA      AGE
```

```
cm-appconfigfiles 2          14s

# kubectl describe configmap cm-appconfigfiles
Name:           cm-appconfigfiles
Namespace:      default
Labels:         <none>
Annotations:   <none>

Data
=====
key-loggingproperties: 1809 bytes
key-serverxml:        1686 bytes
```

查看已创建的 ConfigMap 的详细内容，可以看到两个配置文件的全文：

```
# kubectl get configmap cm-appconfigfiles -o yaml
apiVersion: v1
data:
  key-loggingproperties: "handlers = 1catalina.org.apache.juli.FileHandler,
2localhost.org.apache.juli.FileHandler,
3manager.org.apache.juli.FileHandler, 4host-manager.org.apache.juli.
FileHandler,
      java.util.logging.ConsoleHandler\r\n\r\n.handlers = 1catalina.org.apache.
juli.FileHandler,
      java.util.logging.ConsoleHandler\r\n\r\n1catalina.org.apache.juli.
FileHandler.level
      = FINE\r\n1catalina.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHandler.prefix
      = catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.level =
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
      = ${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHandler.prefix =
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.level
      = FINE\r\n3manager.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandler.prefix
      = manager.\r\n\r\n4host-manager.org.apache.juli.FileHandler.level =
FINE\r\n4host-manager.org.apache.juli.FileHandler.directory
      = ${catalina.base}/logs\r\n4host-manager.org.apache.juli.FileHandler.
prefix =
      host-manager.\r\n\r\njava.util.logging.ConsoleHandler.level = FINE\r\njava.
util.logging.ConsoleHandler.formatter
      = java.util.logging.SimpleFormatter\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].level
      = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
handlers
      = 2localhost.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/manager].level
      = INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager].handlers
```

```

= 3manager.org.apache.juli.FileHandler\r\n\r\norg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/host-manager].level
= INFO\r\norg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/host-manager].handlers
= 4host-manager.org.apache.juli.FileHandler\r\n\r\n"
key-serverxml: |
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
    <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
    <Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
    <Listener className="org.apache.catalina.core.
JreMemoryLeakPreventionListener" />
    <Listener className="org.apache.catalina.mbeans.
GlobalResourcesLifecycleListener" />
    <Listener className="org.apache.catalina.core.
ThreadLocalLeakPreventionListener" />
<GlobalNamingResources>
    <Resource name="UserDatabase" auth="Container"
        type="org.apache.catalina.UserDatabase"
        description="User database that can be updated and saved"
        factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
        pathname="conf/tomcat-users.xml" />
</GlobalNamingResources>

<Service name="Catalina">
    <Connector port="8080" protocol="HTTP/1.1"
        connectionTimeout="20000"
        redirectPort="8443" />
    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
    <Engine name="Catalina" defaultHost="localhost">
        <Realm className="org.apache.catalina.realm.LockOutRealm">
            <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
                resourceName="UserDatabase"/>
        </Realm>
        <Host name="localhost" appBase="webapps"
            unpackWARs="true" autoDeploy="true">
            <Valve className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
                prefix="localhost_access_log" suffix=".txt"
                pattern="%h %l %u \"%r\" %s %b" />
        </Host>
    </Engine>
</Service>
</Server>
kind: ConfigMap

```

```
metadata:  
  creationTimestamp: 2016-07-29T00:52:18Z  
  name: cm-appconfigfiles  
  namespace: default  
  resourceVersion: "85054"  
  selfLink: /api/v1/namespaces/default/configmaps/cm-appconfigfiles  
  uid: b30d5019-5526-11e6-9dcd-000c29dc2102
```

3. ConfigMap 的创建：kubectl 命令行方式

不使用 yaml 文件，直接通过 `kubectl create configmap` 也可以创建 ConfigMap，可以使用参数`--from-file` 或 `--from-literal` 指定内容，并且可以在一行命令中指定多个参数。

(1) 通过`--from-file` 参数从文件中进行创建，可以指定 key 的名称，也可以在一个命令行中创建包含多个 key 的 ConfigMap，语法为：

```
# kubectl create configmap NAME --from-file=[key]=source --from-file=[key]=source
```

(2) 通过`--from-file` 参数从目录中进行创建，该目录下的每个配置文件名都被设置为 key，文件的内容被设置为 value，语法为：

```
# kubectl create configmap NAME --from-file=config-files-dir
```

(3) `--from-literal` 从文本中进行创建，直接将指定的 key#value# 创建为 ConfigMap 的内容，语法为：

```
# kubectl create configmap NAME --from-literal=key1=value1 --from-literal=key2=value2
```

下面对这几种用法举例说明。

例如，当前目录下含有配置文件 `server.xml`，可以创建一个包含该文件内容的 ConfigMap：

```
# kubectl create configmap cm-server.xml --from-file=server.xml  
configmap "cm-server.xml" created
```

```
# kubectl describe configmap cm-server.xml  
Name:         cm-server.xml  
Namespace:    default  
Labels:       <none>  
Annotations:  <none>  
  
Data  
====  
server.xml:   6458 bytes
```

假设 `configfiles` 目录下包含两个配置文件 `server.xml` 和 `logging.properties`，创建一个包含这两个文件内容的 ConfigMap：

```
# kubectl create configmap cm-appconf --from-file=configfiles
configmap "cm-appconf" created

# kubectl describe configmap cm-appconf
Name:           cm-appconf
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
=====
logging.properties:   3354 bytes
server.xml:          6458 bytes
```

使用--from-literal 参数进行创建的示例如下：

```
# kubectl create configmap cm-appenv --from-literal=loglevel=info --from-literal
=appdatadir=/var/data
configmap "cm-appenv" created

# kubectl describe configmap cm-appenv
Name:           cm-appenv
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data
=====
appdatadir:     9 bytes
loglevel:       4 bytes
```

容器应用对 ConfigMap 的使用有以下两种方法。

- (1) 通过环境变量获取 ConfigMap 中的内容。
- (2) 通过 Volume 挂载的方式将 ConfigMap 中的内容挂载为容器内部的文件或目录。

4. ConfigMap 的使用：环境变量方式

以 cm-appvars.yaml 为例：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-appvars
data:
  appLogLevel: info
  appDataDir: /var/data
```

在 Pod “cm-test-pod”的定义中，将 ConfigMap “cm-appvars” 中的内容以环境变量 (APPLOGLEVEL 和 APPDATADIR) 设置为容器内部的环境变量，容器的启动命令将显示这两个环境变量的值 (“env | grep APP”):

```
apiVersion: v1
kind: Pod
metadata:
  name: cm-test-pod
spec:
  containers:
  - name: cm-test
    image: busybox
    command: [ "/bin/sh", "-c", "env | grep APP" ]
    env:
      - name: APPLOGLEVEL          # 定义环境变量名称
        valueFrom:
          configMapKeyRef:
            name: cm-appvars       # 环境变量的值取自 cm-appvars 中:
            key: apploglevel        # key 为 “apploglevel”
      - name: APPDATADIR           # 定义环境变量名称
        valueFrom:
          configMapKeyRef:
            name: cm-appvars       # 环境变量的值取自 cm-appvars 中:
            key: appdatadir         # key 为 “appdatadir”
  restartPolicy: Never
```

使用 kubectl create -f 命令创建该 Pod，由于是测试 Pod，所以该 Pod 在执行完启动命令后将会退出，并且不会被系统自动重启 (restartPolicy=Never):

```
# kubectl create -f cm-test-pod.yaml
pod "cm-test-pod" created
```

使用 kubectl get pods --show-all 查看已经停止的 Pod:

```
# kubectl get pods --show-all
NAME          READY   STATUS    RESTARTS   AGE
cm-test-pod   0/1     Completed   0          8s
```

查看该 Pod 的日志，可以看到启动命令 “env | grep APP” 的执行结果如下:

```
# kubectl logs cm-test-pod
APPDATADIR=/var/data
APPLOGLEVEL=info
```

说明容器内部的环境变量使用 ConfigMap cm-appvars 中的值进行了正确的设置。

5. ConfigMap 的使用: volumeMount 模式

下面 cm-appconfigfiles.yaml 的例子中包含两个配置文件的定义: server.xml 和 logging.properties。

```
cm-appconfigfiles.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-serverxml
data:
  key-serverxml: |
    <?xml version='1.0' encoding='utf-8'?>
    <Server port="8005" shutdown="SHUTDOWN">
      <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
      <Listener className="org.apache.catalina.core.AprLifecycleListener"
      SSLEngine="on" />
      <Listener className="org.apache.catalina.core.
      JreMemoryLeakPreventionListener" />
      <Listener className="org.apache.catalina.mbeans.
      GlobalResourcesLifecycleListener" />
      <Listener className="org.apache.catalina.core.
      ThreadLocalLeakPreventionListener" />
      <GlobalNamingResources>
        <Resource name="UserDatabase" auth="Container"
          type="org.apache.catalina.UserDatabase"
          description="User database that can be updated and saved"
          factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
          pathname="conf/tomcat-users.xml" />
      </GlobalNamingResources>

      <Service name="Catalina">
        <Connector port="8080" protocol="HTTP/1.1"
          connectionTimeout="20000"
          redirectPort="8443" />
        <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
        <Engine name="Catalina" defaultHost="localhost">
          <Realm className="org.apache.catalina.realm.LockOutRealm">
            <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
              resourceName="UserDatabase"/>
          </Realm>
          <Host name="localhost" appBase="webapps"
            unpackWARs="true" autoDeploy="true">
            <Valve className="org.apache.catalina.valves.AccessLogValve"
            directory="logs"
              prefix="localhost_access_log" suffix=".txt"
              pattern="%h %l %u %t "%r" %s %b" />
```

```

        </Host>
    </Engine>
</Service>
</Server>
key-loggingproperties: "handlers
    = 1catalina.org.apache.juli.FileHandler,
2localhost.org.apache.juli.FileHandler,
3manager.org.apache.juli.FileHandler,
4host-manager.org.apache.juli.FileHandler,
    java.util.logging.ConsoleHandler\r\n\r\n.handlers =
1catalina.org.apache.juli.FileHandler,

java.util.logging.ConsoleHandler\r\n\r\n\r\n1catalina.org.apache.juli.FileHandler.level
    = FINE\r\n1catalina.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n1catalina.org.apache.juli.FileHandler.prefix
    = catalina.\r\n\r\n2localhost.org.apache.juli.FileHandler.level =
FINE\r\n2localhost.org.apache.juli.FileHandler.directory
    = ${catalina.base}/logs\r\n2localhost.org.apache.juli.FileHandler.prefix =
localhost.\r\n\r\n3manager.org.apache.juli.FileHandler.level
    = FINE\r\n3manager.org.apache.juli.FileHandler.directory =
${catalina.base}/logs\r\n3manager.org.apache.juli.FileHandler.prefix
    = manager.\r\n\r\n4host-manager.org.apache.juli.FileHandler.level =
FINE\r\n4host-manager.org.apache.juli.FileHandler.directory
    = ${catalina.base}/logs\r\n4host-manager.org.apache.juli.FileHandler.
prefix =
        host-manager.\r\n\r\njava.util.logging.ConsoleHandler.level =
FINE\r\njava.util.logging.ConsoleHandler.formatter
    = java.util.logging.SimpleFormatter\r\n\r\n\r\nnorg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].level
    = INFO\r\nnorg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
handlers
    = 2localhost.org.apache.juli.FileHandler\r\n\r\n\r\nnorg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/manager].level
    = INFO\r\nnorg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/manager].handlers
    = 3manager.org.apache.juli.FileHandler\r\n\r\n\r\nnorg.apache.catalina.core.
ContainerBase.[Catalina].[localhost].[/host-manager].level
    = INFO\r\nnorg.apache.catalina.core.ContainerBase.[Catalina].[localhost].
[/host-manager].handlers
    = 4host-manager.org.apache.juli.FileHandler\r\n\r\n\r\n"

```

在 Pod “cm-test-app”的定义中，将 ConfigMap “cm-appconfigfiles” 中的内容以文件的形式 mount 到容器内部的/configfiles 目录中去。Pod 配置文件 cm-test-app.yaml 的内容如下：

```

apiVersion: v1
kind: Pod
metadata:

```

```

name: cm-test-app
spec:
  containers:
    - name: cm-test-app
      image: kubeguide/tomcat-app:v1
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: serverxml          # 引用 volume 名
          mountPath: /configfiles # 挂载到容器内的目录
  volumes:
    - name: serverxml          # 定义 volume 名
      configMap:
        name: cm-appconfigfiles # 使用 ConfigMap “cm-appconfigfiles”
        items:
          - key: key-serverxml     # key=key-serverxml
            path: server.xml       # value 将 server.xml 文件名进行挂载
          - key: key-loggingproperties # key=key-loggingproperties
            path: logging.properties # value 将 logging.properties 文件名进行挂载

```

创建该 Pod:

```
# kubectl create -f cm-test-app.yaml
pod "cm-test-app" created
```

登录容器，查看到/configfiles 目录下存在 server.xml 和 logging.properties 文件，它们的内容就是 ConfigMap “cm-appconfigfiles” 中两个 key 定义的内容。

```
# kubectl exec -ti cm-test-app -- bash
root@cm-test-app:/# cat /configfiles/server.xml
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
  .....

root@cm-test-app:/# cat /configfiles/logging.properties
handlers = 1catalina.org.apache.juli.AsyncFileHandler,
2localhost.org.apache.juli.AsyncFileHandler,
3manager.org.apache.juli.AsyncFileHandler,
4host-manager.org.apache.juli.AsyncFileHandler, java.util.logging.ConsoleHandler
.....
```

如果在引用 ConfigMap 时不指定 items，则使用 volumeMount 方式在容器内的目录中为每个 item 生成一个文件名为 key 的文件。

Pod 配置文件 cm-test-app.yaml 的内容如下：

```
apiVersion: v1
kind: Pod
metadata:
```

```
name: cm-test-app
spec:
  containers:
    - name: cm-test-app
      image: kubeguide/tomcat-app:v1
      imagePullPolicy: Never
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: serverxml          # 引用 volume 名
          mountPath: /configfiles # 挂载到容器内的目录
  volumes:
    - name: serverxml          # 定义 volume 名
  configMap:
    name: cm-appconfigfiles   # 使用 ConfigMap “cm-appconfigfiles”
```

创建该 Pod:

```
# kubectl create -f cm-test-app.yaml
pod "cm-test-app" created
```

登录容器，查看到/configfiles 目录下存在 key-loggingproperties 和 key-serverxml 文件，文件的名称来自 ConfigMap cm-appconfigfiles 中定义的两个 key 的名称，文件的内容则为 value 的内容：

```
# ls /configfiles
key-loggingproperties  key-serverxml
```

6. 使用 ConfigMap 的限制条件

使用 ConfigMap 的限制条件如下。

- ConfigMap 必须在 Pod 之前创建。
- ConfigMap 也可以定义为属于某个 Namespace。只有处于相同 Namespaces 中的 Pod 可以引用它。
- ConfigMap 中的配额管理还未能实现。
- kubelet 只支持可以被 API Server 管理的 Pod 使用 ConfigMap。kubelet 在本 Node 上通过 --manifest-url 或--config 自动创建的静态 Pod 将无法引用 ConfigMap。
- 在 Pod 对 ConfigMap 进行挂载 (volumeMount) 操作时，容器内部只能挂载为“目录”，无法挂载为“文件”。在挂载到容器内部后，目录中将包含 ConfigMap 定义的每个 item，如果该目录下原先还有其他文件，则容器内的该目录将会被挂载的 ConfigMap 进行覆盖。如果应用程序需要保留原来的其他文件，则需要进行额外的处理。可以通过将

ConfigMap 挂载到容器内部的临时目录，再通过启动脚本将配置文件复制或者链接（cp 或 link 操作）到应用所用的实际配置目录下。

2.4.6 Pod 生命周期和重启策略

Pod 在整个生命周期过程中被系统定义为各种状态，熟悉 Pod 的各种状态对于我们理解如何设置 Pod 的调度策略、重启策略是很有必要的。

Pod 的状态包括以下几种，如表 2.14 所示。

表 2.14 Pod 的状态

状态值	描述
Pending	API Server 已经创建该 Pod，但 Pod 内还有一个或多个容器的镜像没有创建，包括正在下载镜像的过程
Running	Pod 内所有容器均已创建，且至少有一个容器处于运行状态、正在启动状态或正在重启状态
Succeeded	Pod 内所有容器均成功执行退出，且不会再重启
Failed	Pod 内所有容器均已退出，但至少有一个容器退出为失败状态
Unknown	由于某种原因无法获取该 Pod 的状态，可能由于网络通信不畅导致

Pod 的重启策略（**RestartPolicy**）应用于 Pod 内的所有容器，并且仅在 Pod 所处的 Node 上由 kubelet 进行判断和重启操作。当某个容器异常退出或者健康检查（详见下节）失败时，kubelet 将根据 RestartPolicy 的设置来进行相应的操作。

Pod 的重启策略包括 Always、OnFailure 和 Never，默认值为 Always。

- ◎ Always：当容器失效时，由 kubelet 自动重启该容器。
- ◎ OnFailure：当容器终止运行且退出码不为 0 时，由 kubelet 自动重启该容器。
- ◎ Never：不论容器运行状态如何，kubelet 都不会重启该容器。

kubelet 重启失效容器的时间间隔以 sync-frequency 乘以 $2n$ 来计算，例如 1、2、4、8 倍等，最长延时 5 分钟，并且在成功重启后的 10 分钟后重置该时间。

Pod 的重启策略与控制方式息息相关，当前可用于管理 Pod 的控制器包括 ReplicationController、Job、DaemonSet 及直接通过 kubelet 管理（静态 Pod）。每种控制器对 Pod 的重启策略要求如下。

- ◎ RC 和 DaemonSet：必须设置为 Always，需要保证该容器持续运行。
- ◎ Job：OnFailure 或 Never，确保容器执行完成后不再重启。
- ◎ kubelet：在 Pod 失效时自动重启它，不论 RestartPolicy 设置为什么值，并且也不会对 Pod 进行健康检查。

结合 Pod 的状态和重启策略，表 2.15 列出一些常见的状态转换场景。

表 2.15 一些常见的状态转换场景

Pod 包含的容器数	Pod 当前的状态	发 生 事 件	Pod 的结果状态		
			RestartPolicy= Always	RestartPolicy= OnFailure	RestartPolicy= Never
包含 1 个容器	Running	容器成功退出	Running	Succeeded	Succeeded
包含 1 个容器	Running	容器失败退出	Running	Running	Failed
包含两个容器	Running	1 个容器失败退出	Running	Running	Running
包含两个容器	Running	容器被 OOM 杀掉	Running	Running	Failed

2.4.7 Pod 健康检查

对 Pod 的健康状态检查可以通过两类探针来检查：LivenessProbe 和 ReadinessProbe。

- ◎ **LivenessProbe 探针：**用于判断容器是否存活（running 状态），如果 LivenessProbe 探针探测到容器不健康，则 kubelet 将杀掉该容器，并根据容器的重启策略做相应的处理。如果一个容器不包含 LivenessProbe 探针，那么 kubelet 认为该容器的 LivenessProbe 探针返回的值永远是“Success”。
- ◎ **ReadinessProbe：**用于判断容器是否启动完成（ready 状态），可以接收请求。如果 ReadinessProbe 探针检测到失败，则 Pod 的状态将被修改。Endpoint Controller 将从 Service 的 Endpoint 中删除包含该容器所在 Pod 的 Endpoint。

kubelet 定期执行 LivenessProbe 探针来诊断容器的健康状况。LivenessProbe 有以下三种实现方式。

- (1) ExecAction: 在容器内部执行一个命令，如果该命令的返回码为 0，则表明容器健康。

在下面的例子中，通过执行“cat /tmp/health”命令来判断一个容器运行是否正常。而该 Pod 运行之后，在创建/tmp/health 文件的 10 秒之后将删除该文件，而 LivenessProbe 健康检查的初始探测时间（initialDelaySeconds）为 15 秒，探测结果将是 Fail，将导致 kubelet 杀掉该容器并重启它。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
```

```

image: gcr.io/google_containers/busybox
args:
- /bin/sh
- -c
- echo ok > /tmp/health; sleep 10; rm -rf /tmp/health; sleep 600
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/health
  initialDelaySeconds: 15
  timeoutSeconds: 1

```

(2) **TCPSocketAction**: 通过容器的 IP 地址和端口号执行 TCP 检查, 如果能够建立 TCP 连接, 则表明容器健康。

在下面的例子中, 通过与容器内的 localhost:80 建立 TCP 连接进行健康检查。

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-healthcheck
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
  livenessProbe:
    tcpSocket:
      port: 80
    initialDelaySeconds: 30
    timeoutSeconds: 1

```

(3) **HTTPGetAction**: 通过容器的 IP 地址、端口号及路径调用 HTTP Get 方法, 如果响应的状态码大于等于 200 且小于等于 400, 则认为容器状态健康。

在下面的例子中, kubelet 定时发送 HTTP 请求到 localhost:80/_status/healthz 来进行容器应用的健康检查。

```

apiVersion: v1
kind: Pod
metadata:
  name: pod-with-healthcheck
spec:
  containers:
  - name: nginx
    image: nginx

```

```

ports:
- containerPort: 80
livenessProbe:
  httpGet:
    path: /_status/healthz
    port: 80
  initialDelaySeconds: 30
  timeoutSeconds: 1

```

对于每种探测方式，都需要设置 `initialDelaySeconds` 和 `timeoutSeconds` 两个参数，它们的含义分别如下。

- ◎ **`initialDelaySeconds`**: 启动容器后进行首次健康检查的等待时间，单位为秒。
- ◎ **`timeoutSeconds`**: 健康检查发送请求后等待响应的超时时间，单位为秒。当超时发生时，`kubelet` 会认为容器已经无法提供服务，将会重启该容器。

2.4.8 玩转 Pod 调度

在 Kubernetes 系统中，Pod 在大部分场景下都只是容器的载体而已，通常需要通过 RC、Deployment、DaemonSet、Job 等对象来完成 Pod 的调度与自动控制功能。

1. RC、Deployment: 全自动调度

RC 的主要功能之一就是自动部署一个容器应用的多份副本，以及持续监控副本的数量，在集群内始终维持用户指定的副本数量。

根据 `frontend-controller.yaml` 配置，用户需要创建 3 个 `kubeguide/guestbook-php-frontend` 应用的副本，在将该定义发送给 Kubernetes 之后，系统将在集群中合适的 Node 上创建 3 个 Pod，并始终维持 3 个副本的数量。

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:

```

```

      name: frontend
spec:
  containers:
    - name: frontend
      image: kubeguide/guestbook-php-frontend
      env:
        - name: GET_HOSTS_FROM
          value: env
      ports:
        - containerPort: 80

```

在调度策略上，除了使用系统内置的调度算法选择合适的 Node 进行调度，也可以在 Pod 的定义中使用 NodeSelector 或 NodeAffinity 来指定满足条件的 Node 进行调度，下面我们分别进行说明。

1) NodeSelector：定向调度

Kubernetes Master 上的 Scheduler 服务（kube-scheduler 进程）负责实现 Pod 的调度，整个调度过程通过执行一系列复杂的算法，最终为每个 Pod 计算出一个最佳的目标节点，这一过程是自动完成的，通常我们无法知道 Pod 最终会被调度到哪个节点上。在实际情况中，也可能需要将 Pod 调度到指定的一些 Node 上，可以通过 Node 的标签（Label）和 Pod 的 nodeSelector 属性相匹配，来达到上述目的。

(1) 首先通过 `kubectl label` 命令给目标 Node 打上一些标签：

```
kubectl label nodes <node-name> <label-key>=<label-value>
```

这里，我们为 `k8s-node-1` 节点打上一个 `zone=north` 的标签，表明它是“北方”的一个节点：

```
$ kubectl label nodes k8s-node-1 zone=north
NAME           LABELS                                     STATUS
k8s-node-1     kubernetes.io/hostname=k8s-node-1,zone=north  Ready
```

上述命令行操作也可以通过修改资源定义文件的方式，并执行 `kubectl replace -f xxx.yaml` 命令来完成。

(2) 然后，在 Pod 的定义中加上 nodeSelector 的设置，以 `redis-master-controller.yaml` 为例：

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master

```

```

template:
  metadata:
    labels:
      name: redis-master.
  spec:
    containers:
      - name: master
        image: kubeguide/redis-master
        ports:
          - containerPort: 6379
    nodeSelector:
      zone: north

```

运行 `kubectl create -f` 命令创建 Pod，scheduler 就会将该 Pod 调度到拥有 `zone=north` 标签的 Node 上。

使用 `kubectl get pods -o wide` 命令可以验证 Pod 所在的 Node：

```
# kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE     NODE
redis-master-f0rqj   1/1    Running   0          19s   k8s-node-1
```

如果我们给多个 Node 都定义了相同的标签（例如 `zone=north`），则 scheduler 将会根据调度算法从这组 Node 中挑选一个可用的 Node 进行 Pod 调度。

通过基于 Node 标签的调度方式，我们可以把集群中具有不同特点的 Node 贴上不同的标签，例如“`role=frontend`”“`role=backend`”“`role=database`”等标签，在部署应用时就可以根据应用的需求设置 `NodeSelector` 来进行指定 Node 范围的调度。

需要注意的是，如果我们指定了 Pod 的 `nodeSelector` 条件，且集群中不存在包含相应标签的 Node，则即使集群中还有其他可供使用的 Node，这个 Pod 也无法被成功调度。

2) NodeAffinity：亲和性调度

`NodeAffinity` 意为 Node 亲和性的调度策略，是将来替换 `NodeSelector` 的下一代调度策略。由于 `NodeSelector` 通过 Node 的 Label 进行精确匹配，所以 `NodeAffinity` 增加了 `In`、`NotIn`、`Exists`、`DoesNotExist`、`Gt`、`Lt` 等操作符来选择 Node，能够使调度策略更加灵活。同时，在 `NodeAffinity` 中将增加一些信息来设置亲和性调度策略。

- ◎ `RequiredDuringSchedulingRequiredDuringExecution`: 类似于 `NodeSelector`，但在 Node 不满足条件时，系统将从该 Node 上移除之前调度上的 Pod。
- ◎ `RequiredDuringSchedulingIgnoredDuringExecution`: 与第 1 个 `RequiredDuringSchedulingRequiredDuringExecution` 的作用相似，区别是在 Node 不满足条件时，系统不一定从该 Node 上移除之前调度上的 Pod。

- PreferedDuringSchedulingIgnoredDuringExecution: 指定在满足调度条件的 Node 中，哪些 Node 应更优先地进行调度。同时在 Node 不满足条件时，系统不一定从该 Node 上移除之前调度上的 Pod。

在当前的 Alpha 版本中，需要在 Pod 的 metadata.annotations 中设置 NodeAffinity 的内容：

```
apiVersion: v1
kind: Pod
metadata:
  name: with-labels
  annotations:
    scheduler.alpha.kubernetes.io/affinity: >
    {
      "nodeAffinity": {
        "requiredDuringSchedulingIgnoredDuringExecution": {
          "nodeSelectorTerms": [
            {
              "matchExpressions": [
                {
                  "key": "kubernetes.io/e2e-az-name",
                  "operator": "In",
                  "values": ["e2e-az1", "e2e-az2"]
                }
              ]
            }
          ]
        }
      }
    }
  another-annotation-key: another-annotation-value
spec:
  containers:
  - name: with-labels
    image: gcr.io/google_containers/pause:2.0
```

这里 NodeAffinity 的设置说明只有 Node 的 Label 中包含 key= kubernetes.io/e2e-az-name，并且其 value 为 “e2e-az1” 或 “e2e-az2” 时，才能成为该 Pod 的调度目标。其中操作符为 In，代表“或”运算，其他操作符包括 NotIn（不属于）、Exists（存在一个条件）、DoesNotExist（不存在）、Gt（大于）、Lt（小于）。

如果同时设置了 NodeSelector 和 NodeAffinity，则系统将需要同时满足两者的设置才能进行调度。

在未来的 Kubernetes 版本中，还将加入 Pod Affinity 的设置，用于控制当调度 Pod 到某个特定的 Node 上时，判断是否有其他 Pod 正在该 Node 上运行，即通过其他的相关 Pod 进行调度，

而不仅仅通过 Node 本身的标签进行调度。

2. DaemonSet：特定场景调度

DaemonSet 是 Kubernetes 1.2 版本新增的一种资源对象，用于管理在集群中每个 Node 上仅运行一份 Pod 的副本实例，如图 2.10 所示。

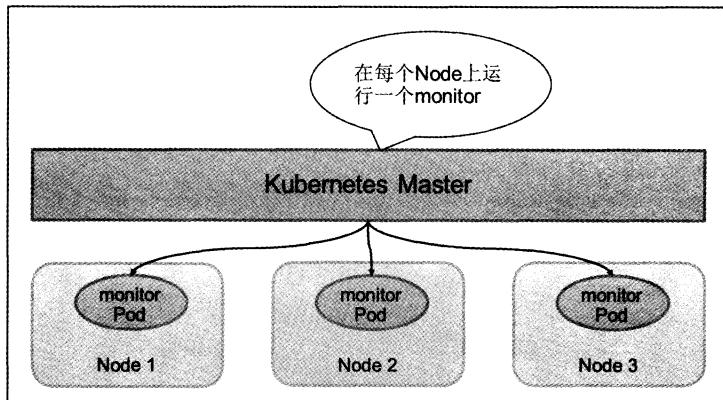


图 2.10 DaemonSet 示例

这种用法适合一些有这种需求的应用。

- ◎ 在每个 Node 上运行一个 GlusterFS 存储或者 Ceph 存储的 daemon 进程。
- ◎ 在每个 Node 上运行一个日志采集程序，例如 fluentd 或者 logstash。
- ◎ 在每个 Node 上运行一个健康程序，采集该 Node 的运行性能数据，例如 Prometheus Node Exporter、collectd、New Relic agent 或者 Ganglia gmond 等。

DaemonSet 的 Pod 调度策略与 RC 类似，除了使用系统内置的算法在每台 Node 上进行调度，也可以在 Pod 的定义中使用 NodeSelector 或 NodeAffinity 来指定满足条件的 Node 范围进行调度。

下面的例子定义为在每台 Node 上启动一个 fluentd 容器，配置文件 fluentd-ds.yaml 的内容如下，其中挂载了物理机的两个目录 “/var/log” 和 “/var/lib/docker/containers”：

```

apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd-cloud-logging
  namespace: kube-system
  labels:
    k8s-app: fluentd-cloud-logging
spec:
  template:
    
```

```

metadata:
  namespace: kube-system
  labels:
    k8s-app: fluentd-cloud-logging
spec:
  containers:
  - name: fluentd-cloud-logging
    image: gcr.io/google_containers/fluentd-elasticsearch:1.17
    resources:
      limits:
        cpu: 100m
        memory: 200Mi
    env:
      - name: FLUENTD_ARGS
        value: -q
    volumeMounts:
      - name: varlog
        mountPath: /var/log
        readOnly: false
      - name: containers
        mountPath: /var/lib/docker/containers
        readOnly: false
  volumes:
    - name: containers
      hostPath:
        path: /var/lib/docker/containers
    - name: varlog
      hostPath:
        path: /var/log

```

使用 kubectl create 命令创建该 DaemonSet:

```
# kubectl create -f fluentd-ds.yaml
daemonset "fluentd-cloud-logging" created
```

查看创建好的 DaemonSet 和 Pod, 可以看到在每个 Node 上都创建了一个 Pod:

```
# kubectl get daemonset --namespace=kube-system
NAME           DESIRED   CURRENT   NODE-SELECTOR   AGE
fluentd-cloud-logging   2         2         <none>       3s

# kubectl get pods --namespace=kube-system
NAME                  READY   STATUS    RESTARTS   AGE
fluentd-cloud-logging-7tw9z   1/1     Running   0          1h
fluentd-cloud-logging-aqdn1   1/1     Running   0          1h
```

3. Job：批处理调度

Kubernetes 从 1.2 版本开始支持批处理类型的应用，我们可以通过 Kubernetes Job 资源对象来定义并启动一个批处理任务。批处理任务通常并行（或者串行）启动多个计算进程去处理一批工作项（work item），处理完成后，整个批处理任务结束。按照批处理任务实现方式的不同，批处理任务可以分为如图 2.11 所示的几种模式。

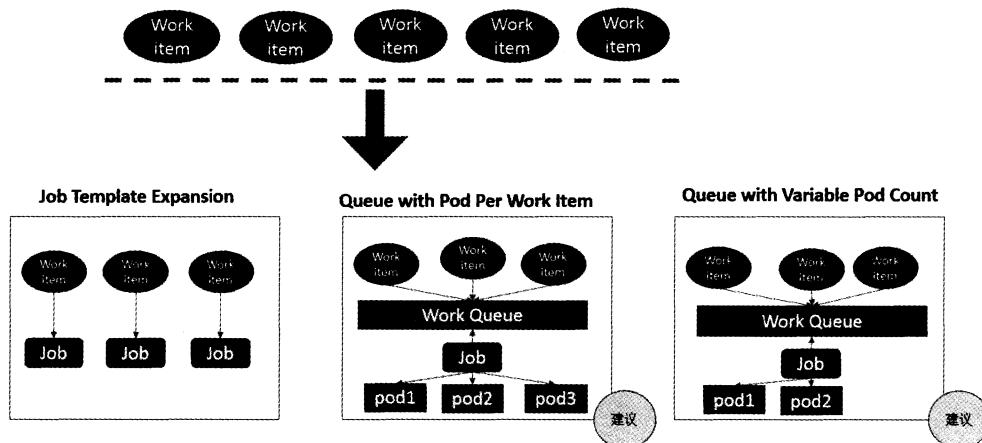


图 2.11 批处理任务的几种模式

- ◎ **Job Template Expansion** 模式：一个 Job 对象对应一个待处理的 Work item，有几个 Work item 就产生几个独立的 Job，通常适合 Work item 数量少、每个 Work item 要处理的数据量比较大的场景，比如有一个 100GB 的文件作为一个 Work item，总共 10 个文件需要处理。
- ◎ **Queue with Pod Per Work Item** 模式：采用一个任务队列存放 Work item，一个 Job 对象作为消费者去完成这些 Work item，在这种模式下，Job 会启动 N 个 Pod，每个 Pod 对应一个 Work item。
- ◎ **Queue with Variable Pod Count** 模式：也是采用一个任务队列存放 Work item，一个 Job 对象作为消费者去完成这些 Work item，但与上面的模式不同，Job 启动的 Pod 数量是可变的。

还有一种被称为 Single Job with Static Work Assignment 的模式，也是一个 Job 产生多个 Pod 的模式，但它采用程序静态方式分配任务项，而不是采用队列模式进行动态分配。

如表 2.16 所示是这几种模式的一个对比。

模 式 名 称	是否是一个 Job	Pod 的数量少于 Work item	用户程序是否要做相应的修改	Kubernetes 是否支持
Job Template Expansion	/	/	是	是
Queue with Pod Per Work Item	是	/	有时候需要	是
Queue with Variable Pod Count	是	/	/	是
Single Job with Static Work Assignment	是	/	是	/

考虑到批处理的并行问题，Kubernetes 将 Job 分以下三种类型。

1) Non-parallel Jobs

通常一个 Job 只启动一个 Pod，则除非 Pod 异常，才会重启该 Pod，一旦此 Pod 正常结束，Job 将结束。

2) Parallel Jobs with a fixed completion count

并行 Job 会启动多个 Pod，此时需要设定 Job 的.spec.completions 参数为一个正数，当正常结束的 Pod 数量达到此参数设定的值后，Job 结束。此外，Job 的.spec.parallelism 参数用来控制并行度，即同时启动几个 Job 来处理 Work Item。

3) Parallel Jobs with a work queue

任务队列方式的并行 Job 需要一个独立的 Queue，Work item 都在一个 Queue 中存放，不能设置 Job 的.spec.completions 参数，此时 Job 有以下一些特性。

- ◎ 每个 Pod 能独立判断和决定是否还有任务项需要处理。
- ◎ 如果某个 Pod 正常结束，则 Job 不会再启动新的 Pod。
- ◎ 如果一个 Pod 成功结束，则此时应该不存在其他 Pod 还在干活的情况，它们应该都处于即将结束、退出的状态。
- ◎ 如果所有 Pod 都结束了，且至少有一个 Pod 成功结束，则整个 Job 算是成功结束。

下面我们分别说说常见的三种批处理模型在 Kubernetes 中的例子。

首先是 Job Template Expansion 模式，由于这种模式下每个 Work item 对应一个 Job 实例，所以这种模式首先定义一个 Job 模板，模板里主要的参数是 Work item 的标识，因为每个 Job 处理不同的 Work item。如下所示的 Job 模板（文件名为 job.yaml.txt）中的\$ITEM 可以作为任务项的标识：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: process-item-$ITEM
  labels:
    jobgroup: jobexample
```

```
spec:  
  template:  
    metadata:  
      name: jobexample  
      labels:  
        jobgroup: jobexample  
    spec:  
      containers:  
      - name: c  
        image: busybox  
        command: ["sh", "-c", "echo Processing item $ITEM && sleep 5"]  
      restartPolicy: Never
```

通过下面的操作，生成 3 个对应的 Job 定义文件并创建 Job：

```
# for i in apple banana cherry  
> do  
>   cat job.yaml.txt | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml  
> done  
# ls jobs  
job-apple.yaml job-banana.yaml job-cherry.yaml  
# kubectl create -f jobs  
job "process-item-apple" created  
job "process-item-banana" created  
job "process-item-cherry" created
```

观察 Job 的运行情况：

```
# kubectl get jobs -l jobgroup=jobexample  
NAME          DESIRED   SUCCESSFUL   AGE  
process-item-apple   1         1           4m  
process-item-banana   1         1           4m  
process-item-cherry   1         1           4m
```

其次，我们看看 Queue with Pod Per Work Item 模式，在这种模式下需要一个任务队列存放 Work item，比如 RabbitMQ，客户端程序先把要处理的任务变成 Work item 放入到任务队列，然后编写 Worker 程序并打包镜像并定义成为 Job 中的 Work Pod，Worker 程序的实现逻辑是从任务队列中拉取一个 Work item 并处理，处理完成后即结束进程，图 2.12 给出了并行度为 2 的一个 Demo 示意图。

最后，我们再看看 Queue with Variable Pod Count 模式，如图 2.13 所示，由于这种模式下，Worker 程序需要知道队列中是否还有等待处理的 Work item，如果有就取出来并处理，否则就认为所有工作完成并结束进程，所以任务队列通常要采用 Redis 或者数据库来实现。

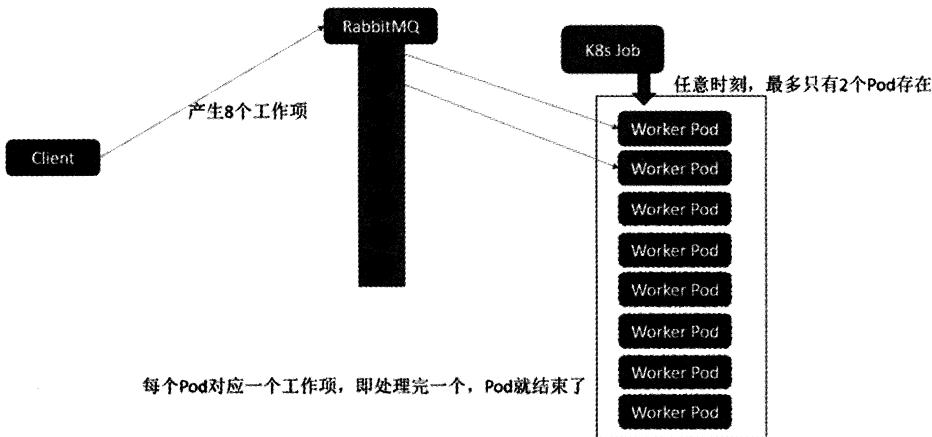


图 2.12 Queue with Pod Per Work Item 示例

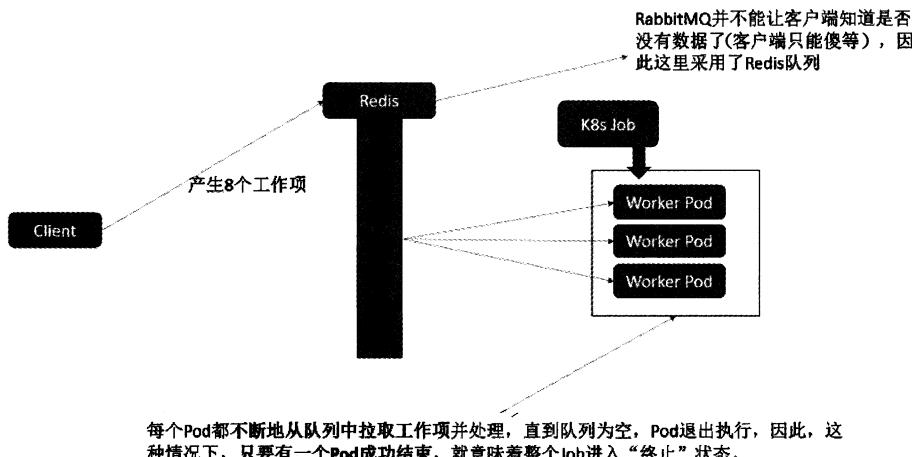


图 2.13 Queue with Variable Pod Count 示例

Kubernetes 对 Job 的支持还处于初级阶段，类似 Linux Cron 的定时任务也还没时间，计划在 Kubernetes 1.4 中实现。此外，更为复杂的流程类的批处理框架也还没有考虑，但随着 Kubernetes 生态圈的不断发展和壮大，相信 Kubernetes 在批处理方面也会有更多的规划。

2.4.9 Pod 的扩容和缩容

在实际生产系统中，我们经常会遇到某个服务需要扩容的场景，也可能会遇到由于资源紧张或者工作负载降低而需要减少服务实例数量的场景。此时我们可以利用 RC 的 Scale 机制来完

成这些工作。以 redis-slave RC 为例，已定义的最初副本数量为 2，通过 kubectl scale 命令可以将 redis-slave RC 控制的 Pod 副本数量从初始的 2 更新为 3：

```
$ kubectl scale rc redis-slave --replicas=3
replicationcontroller "redis-slave" scaled
```

执行 kubectl get pods 命令来验证 Pod 的副本数量增加到 3：

```
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
redis-slave-4na2n  1/1     Running   0          1h
redis-slave-92u3k  1/1     Running   0          1h
redis-slave-palab  1/1     Running   0          2m
```

将--replicas 设置为比当前 Pod 副本数量更小的数字，系统将会“杀掉”一些运行中的 Pod，以实现应用集群缩容：

```
$ kubectl scale rc redis-slave --replicas=1
replicationcontroller "redis-slave" scaled
```

```
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
redis-slave-4na2n  1/1     Running   0          1h
```

除了可以手工通过 kubectl scale 命令完成 Pod 的扩容和缩容操作，Kubernetes v1.1 版本新增了名为 Horizontal Pod Autoscaler (HPA) 的控制器，用于实现基于 CPU 使用率进行自动 Pod 扩缩容的功能。HPA 控制器基于 Master 的 kube-controller-manager 服务启动参数 --horizontal-pod-autoscaler-sync-period 定义的时长（默认为 30 秒），周期性地监测目标 Pod 的 CPU 使用率，并在满足条件时对 ReplicationController 或 Deployment 中的 Pod 副本数量进行调整，以符合用户定义的平均 Pod CPU 使用率。Pod CPU 使用率来源于 heapster 组件，所以需要预先安装好 heapster。

创建 HPA 时可以使用 kubectl autoscale 命令进行快速创建或者使用 yaml 配置文件进行创建。在创建 HPA 之前，需要已经存在一个 RC 或 Deployment 对象，并且该 RC 或 Deployment 中的 Pod 必须定义 resources.requests.cpu 的资源请求值，如果不设置该值，则 heapster 将无法采集到该 Pod 的 CPU 使用情况，会导致 HPA 无法正常工作。

下面通过给一个 RC 设置 HPA，然后使用一个客户端对其进行压力测试，对 HPA 的用法进行示例。

以 php-apache 的 RC 为例，设置 cpu request 为 200m，未设置 limit 上限的值：

```
php-apache-rc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
```

```

name: php-apache
spec:
  replicas: 1
  template:
    metadata:
      name: php-apache
      labels:
        app: php-apache
    spec:
      containers:
        - name: php-apache
          image: gcr.io/google_containers/hpa-example
      resources:
        requests:
          cpu: 200m
      ports:
        - containerPort: 80

```

kubectl create -f php-apache-rc.yaml
replicationcontroller "php-apache" created

再创建一个 php-apache 的 Service，供客户端访问：

```

php-apache-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: php-apache
spec:
  ports:
    - port: 80
  selector:
    app: php-apache

```

kubectl create -f php-apache-svc.yaml
service "php-apache" created

接下来为 RC “php-apache” 创建一个 HPA 控制器，在 1 和 10 之间调整 Pod 的副本数量，以使得平均 Pod CPU 使用率维持在 50%。

使用 kubectl autoscale 命令进行创建：

```
# kubectl autoscale rc php-apache --min=1 --max=10 --cpu-percent=50
```

或者通过 yaml 配置文件来创建 HPA，需要在 scaleTargetRef 字段指定需要管理的 RC 或 Deployment 的名字，然后设置 minReplicas、maxReplicas 和 targetCPUUtilizationPercentage 参数：

```

hpa-php-apache.yaml
apiVersion: autoscaling/v1

```

```
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: v1
    kind: ReplicationController
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50

# kubectl create -f hpa-php-apache.yaml
horizontalpodautoscaler "php-apache" created
```

查看已创建的 HPA：

```
# kubectl get hpa
NAME          REFERENCE          TARGET      CURRENT   MINPODS
MAXPODS   AGE
php-apache   ReplicationController/php-apache   50%       0%       1       10      1m
```

然后，我们创建一个 busybox Pod，用于对 php-apache 服务发起压力测试的请求：

```
busybox-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
  - name: busybox
    image: busybox
    command: [ "sleep", "3600" ]
```

```
# kubectl create -f busybox-pod.yaml
pod "busybox" created
```

登录 busybox 容器，执行一个无限循环的 wget 命令来访问 php-apache 服务：

```
# while true; do wget -q -O- http://php-apache > /dev/null; done
```

注意这里 wget 的目的地 URL 地址是 Service 的名称 “php-apache”，这要求 DNS 服务正常工作，也可以使用 Service 的虚拟 ClusterIP 地址对其进行访问，例如 http://169.169.122.145：

```
# kubectl exec -ti busybox -- sh
/ # while true; do wget -q -O- http://php-apache > /dev/null; done
```

等待一段时间后，观察 HPA 控制器搜集到的 Pod CPU 使用率：

```
# kubectl get hpa
```

NAME	REFERENCE	TARGET	CURRENT	MINPODS	MAXPODS	AGE
php-apache	ReplicationController/php-apache	50%	3068%	1	10	3m

再过一会儿，查看 RC php-apache 副本数量的变化：

```
# kubectl get rc
NAME      DESIRED   CURRENT   AGE
php-apache 10        10        23m
```

可以看到 HPA 已经根据 Pod 的 CPU 使用率的提高对 RC 进行了自动扩容，Pod 副本数量变成了 10 个。这个过程如图 2.14 所示。

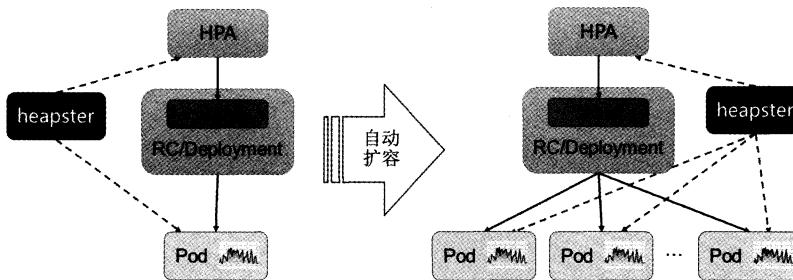


图 2.14 HPA 自动扩容

最后，我们停止压力测试，在 busybox 的控制台输入 Ctrl+C，停止无限循环操作。

等待一段时间，观察 HPA 的变化：

```
# kubectl get hpa
NAME      REFERENCE          TARGET      CURRENT      MINPODS      MAXPODS      AGE
php-apache  ReplicationController/php-apache  50%       3%         1           10          20m
```

再次查看 RC 的副本数量：

```
NAME      DESIRED   CURRENT   AGE
php-apache 1        1        26m
```

可以看到 HPA 根据 Pod CPU 使用率的降低对副本数量进行了缩容操作，Pod 副本数量变成了 1 个。

当前 HPA 还只支持将 CPU 使用率作为 Pod 副本扩容缩容的触发条件，在将来的版本中，将会支持应用相关的指标例如 QPS（queries per second）或平均响应时间作为触发条件。

2.4.10 Pod 的滚动升级

下面我们说说 Pod 的升级问题。

当集群中的某个服务需要升级时，我们需要停止目前与该服务相关的所有 Pod，然后重新

拉取镜像并启动。如果集群规模比较大，则这个工作就变成了一个挑战，而且先全部停止然后逐步升级的方式会导致较长时间的服务不可用。Kubernetes 提供了 rolling-update（滚动升级）功能来解决上述问题。

滚动升级通过执行 kubectl rolling-update 命令一键完成，该命令创建了一个新的 RC，然后自动控制旧的 RC 中的 Pod 副本的数量逐渐减少到 0，同时新的 RC 中的 Pod 副本的数量从 0 逐步增加到目标值，最终实现了 Pod 的升级。需要注意的是，系统要求新的 RC 需要与旧的 RC 在相同的命名空间（Namespace）内，即不能把别人的资产偷偷转移到自家名下。滚动升级的过程如图 2.15 所示。

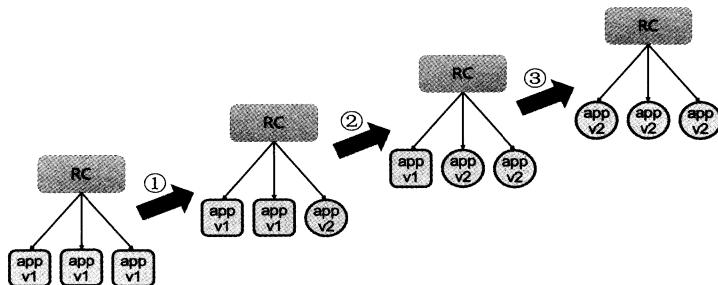


图 2.15 Pod 的滚动升级

以 redis-master 为例，假设当前运行的 redis-master Pod 是 1.0 版本，则现在需要升级到 2.0 版本。

创建 redis-master-controller-v2.yaml 的配置文件如下：

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master-v2
  labels:
    name: redis-master
    version: v2
spec:
  replicas: 1
  selector:
    name: redis-master
    version: v2
  template:
    metadata:
      labels:
        name: redis-master
        version: v2
    spec:
      containers:

```

```

- name: master
  image: kubeguide/redis-master:2.0
  ports:
    - containerPort: 6379

```

在配置文件中需要注意以下几点。

- (1) RC 的名字 (name) 不能与旧的 RC 的名字相同。
- (2) 在 selector 中应至少有一个 Label 与旧的 RC 的 Label 不同, 以标识其为新的 RC。本例中新增了一个名为 version 的 Label, 以与旧的 RC 进行区分。

运行 `kubectl rolling-update` 命令完成 Pod 的滚动升级:

```
kubectl rolling-update redis-master -f redis-master-controller-v2.yaml
```

`kubectl` 的执行过程如下:

```

Creating redis-master-v2
At beginning of loop: redis-master replicas: 2, redis-master-v2 replicas: 1
Updating redis-master replicas: 2, redis-master-v2 replicas: 1
At end of loop: redis-master replicas: 2, redis-master-v2 replicas: 1
At beginning of loop: redis-master replicas: 1, redis-master-v2 replicas: 2
Updating redis-master replicas: 1, redis-master-v2 replicas: 2
At end of loop: redis-master replicas: 1, redis-master-v2 replicas: 2
At beginning of loop: redis-master replicas: 0, redis-master-v2 replicas: 3
Updating redis-master replicas: 0, redis-master-v2 replicas: 3
At end of loop: redis-master replicas: 0, redis-master-v2 replicas: 3
Update succeeded. Deleting redis-master
redis-master-v2

```

等所有新的 Pod 启动完成后, 旧的 Pod 也被全部销毁, 这样就完成了容器集群的更新工作。

另一种方法是不使用配置文件, 直接用 `kubectl rolling-update` 命令, 加上`--image` 参数指定新版镜像名称来完成 Pod 的滚动升级:

```
kubectl rolling-update redis-master --image=redis-master:2.0
```

与使用配置文件的方式不同, 执行的结果是旧的 RC 被删除, 新的 RC 仍将使用旧的 RC 的名字。

`kubectl` 的执行过程如下:

```

Creating redis-master-ea866a5d2c08588c3375b86fb253db75
At beginning of loop: redis-master replicas: 2, redis-master-ea866a5d2c08588c
3375b86fb253db75 replicas: 1
Updating redis-master replicas: 2, redis-master-ea866a5d2c08588c3375b86fb253db
75 replicas: 1
At end of loop: redis-master replicas: 2, redis-master-ea866a5d2c08588c3375b86fb
253db75 replicas: 1
At beginning of loop: redis-master replicas: 1, redis-master-ea866a5d2c08588c

```

```
3375b86fb253db75 replicas: 2
    Updating redis-master replicas: 1, redis-master-ea866a5d2c08588c3375b86fb
253db75 replicas: 2
    At end of loop: redis-master replicas: 1, redis-master-ea866a5d2c08588c3375b86fb
253db75 replicas: 2
    At beginning of loop: redis-master replicas: 0, redis-master-ea866a5d2c08588c
3375b86fb253db75 replicas: 3
    Updating redis-master replicas: 0, redis-master-ea866a5d2c08588c3375b86fb253db
75 replicas: 3
    At end of loop: redis-master replicas: 0, redis-master-ea866a5d2c08588c3375b86fb
253db75 replicas: 3
    Update succeeded. Deleting old controller: redis-master
    Renaming redis-master-ea866a5d2c08588c3375b86fb253db75 to redis-master
    redis-master
```

可以看到，`kubectl` 通过新建一个新版本 Pod，停掉一个旧版本 Pod，逐步迭代来完成整个 RC 的更新。

更新完成后，查看 RC：

```
$ kubectl get rc
CONTROLLER      CONTAINER(S)        IMAGE(S)          SELECTOR          REPLICAS
redis-master     master            kubeguide/redis-master:2.0   deployment=
ea866a5d2c08588c3375b86fb253db75, name=redis-master, version=v1   3
```

可以看到，`kubectl` 给 RC 增加了一个 key 为“deployment”的 Label（这个 key 的名字可通过`--deployment-label-key` 参数进行修改），Label 的值是 RC 的内容进行 Hash 计算后的值，相当于签名，这样就能很方便地比较 RC 里的 Image 名字及其他信息是否发生了变化，它的具体作用可以参见第 6 章的源码分析。

如果在更新过程中发现配置有误，则用户可以中断更新操作，并通过执行 `kubectl rolling-update--rollback` 完成 Pod 版本的回滚：

```
$ kubectl rolling-update redis-master --image=kubeguide/redis-master:2.0 --rollback
Found existing update in progress (redis-master-fefd9752aa5883ca4d53013a7b
583967), resuming.
Found desired replicas. Continuing update with existing controller redis-master.
At beginning of loop: redis-master-fefd9752aa5883ca4d53013a7b583967 replicas:
0, redis-master replicas: 3
Updating redis-master-fefd9752aa5883ca4d53013a7b583967 replicas: 0, redis-master
replicas: 3
At end of loop: redis-master-fefd9752aa5883ca4d53013a7b583967 replicas: 0,
redis-master replicas: 3
Update succeeded. Deleting redis-master-fefd9752aa5883ca4d53013a7b583967
redis-master
```

到此，可以看到 Pod 恢复到更新前的版本了。

2.5 深入掌握 Service

Service 是 Kubernetes 最核心的概念，通过创建 Service，可以为一组具有相同功能的容器应用提供一个统一的入口地址，并且将请求进行负载分发到后端的各个容器应用上。本节对 Service 的使用进行详细说明，包括 Service 的负载均衡、外网访问、DNS 服务的搭建、Ingress 7 层路由机制等。

2.5.1 Service 定义详解

yaml 格式的 Service 定义文件的完整内容如下：

```
apiVersion: v1          // Required
kind: Service           // Required
metadata:               // Required
  name: string          // Required
  namespace: string      // Required
  labels:
    - name: string
  annotations:
    - name: string
spec:                   // Required
  selector: []           // Required
  type: string           // Required
  clusterIP: string
  sessionAffinity: string
  ports:
    - name: string
      protocol: string
      port: int
      targetPort: int
      nodePort: int
  status:
    loadBalancer:
      ingress:
        ip: string
        hostname: string
```

对各属性的说明如表 2.17 所示。

表 2.17 对 Service 的定义文件模板的各属性的说明

属性名称	取值类型	是否必选	取值说明
version	string	Required	v1
kind	string	Required	Service
metadata	object	Required	元数据
metadata.name	string	Required	Service 名称，需符合 RFC 1035 规范
metadata.namespace	string	Required	命名空间，不指定系统时将使用名为“default”的命名空间
metadata.labels[]	list		自定义标签属性列表
metadata.annotation[]	list		自定义注解属性列表
spec	object	Required	详细描述
spec.selector[]	list	Required	Label Selector 配置，将选择具有指定 Label 标签的 Pod 作为管理范围
spec.type	string	Required	Service 的类型，指定 Service 的访问方式，默认为 ClusterIP。 ClusterIP：虚拟的服务 IP 地址，该地址用于 Kubernetes 集群内部的 Pod 访问，在 Node 上 kube-proxy 通过设置的 iptables 规则进行转发。 NodePort：使用宿主机的端口，使能够访问各 Node 的外部客户端通过 Node 的 IP 地址和端口号就能访问服务。 LoadBalancer：使用外接负载均衡器完成到服务的负载分发，需要在 spec.status.loadBalancer 字段指定外部负载均衡器的 IP 地址，并同时定义 nodePort 和 clusterIP，用于公有云环境
spec.clusterIP	string		虚拟服务 IP 地址，当 type=ClusterIP 时，如果不指定，则系统进行自动分配，也可以手工指定；当 type=LoadBalancer 时，则需要指定
spec.sessionAffinity	string		是否支持 Session，可选值为 ClientIP，默认为空。 ClientIP：表示将同一个客户端（根据客户端的 IP 地址决定）的访问请求都转发到同一个后端 Pod
spec.ports[]	list		Service 需要暴露的端口列表
spec.ports[].name	string		端口名称
spec.ports[].protocol	string		端口协议，支持 TCP 和 UDP，默认为 TCP
spec.ports[].port	int		服务监听的端口号
spec.ports[].targetPort	int		需要转发到后端 Pod 的端口号
spec.ports[].nodePort	int		当 spec.type=NodePort 时，指定映射到物理机的端口号
Status	object		当 spec.type=LoadBalancer 时，设置外部负载均衡器的地址，用于公有云环境
status.loadBalancer	object		外部负载均衡器
status.loadBalancer.ingress	object		外部负载均衡器
status.loadBalancer.ingress.ip	string		外部负载均衡器的 IP 地址
status.loadBalancer.ingress.hostname	string		外部负载均衡器的主机名

2.5.2 Service 基本用法

一般来说，对外提供服务的应用程序需要通过某种机制来实现，对于容器应用最简便的方式就是通过 TCP/IP 机制及监听 IP 和端口号来实现。例如，我们定义一个提供 Web 服务的 RC，由两个 tomcat 容器副本组成，每个容器通过 containerPort 设置提供服务的端口号为 8080：

```
webapp-rc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: webapp
spec:
  replicas: 2
  template:
    metadata:
      name: webapp
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: tomcat
        ports:
          -containerPort: 80
```

创建该 RC：

```
# kubectl create -f webapp-rc.yaml
replicationcontroller "webapp" created
```

获取 Pod 的 IP 地址：

```
# kubectl get pods -l app=webapp -o yaml | grep podIP
  podIP: 172.17.1.4
  podIP: 172.17.1.3
```

访问这两个 Pod 提供的 Tomcat 服务：

```
# curl 172.17.1.3:8080
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
  .....
# curl 172.17.1.4:8080
<!DOCTYPE html>
<html lang="en">
  <head>
```

```

<meta charset="UTF-8" />
<title>Apache Tomcat/8.0.35</title>
.....

```

直接通过 Pod 的 IP 地址和端口号可以访问容器应用，但是 Pod 的 IP 地址是不可靠的，例如当 Pod 所在的 Node 发生故障时，Pod 将被 Kubernetes 重新调度到另一台 Node 进行启动，Pod 的 IP 地址将发生变化。更重要的是，如果容器应用本身是分布式的部署方式，通过多个实例共同提供服务，就需要在这些实例的前端设置一个负载均衡器来实现请求的分发。Kubernetes 中的 Service 就是设计出来用于解决这些问题的核心组件。

以前面创建的 webapp 应用为例，为了让客户端应用能够访问到两个 Tomcat Pod 实例，需要创建一个 Service 来提供服务。Kubernetes 提供了一种快速的方法，即通过 kubectl expose 命令来创建 Service：

```
# kubectl expose rc webapp
service "webapp" exposed
```

查看新创建的 Service，可以看到系统为它分配了一个虚拟的 IP 地址（ClusterIP），而 Service 所需的端口号则从 Pod 中的 containerPort 复制而来：

```
# kubectl get svc
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
webapp    169.169.235.79   <none>          8080/TCP    3s
```

接下来，我们就可以通过 Service 的 IP 地址和 Service 的端口号访问该 Service 了：

```
# curl 169.169.235.79:8080
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
.....
```

这里，对 Service 地址 169.169.235.79:8080 的访问被自动负载分发到了后端两个 Pod 之一：172.17.1.3:8080 或 172.17.1.4:8080。

除了使用 kubectl expose 命令创建 Service，我们也可以通过配置文件定义 Service，再通过 kubectl create 命令进行创建。例如对于前面的 webapp 应用，我们可以设置一个 Service，代码如下：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
```

```

- port: 8081
  targetPort: 8080
selector:
  app: webapp

```

Service 定义中的关键字段是 ports 和 selector。本例中 ports 定义部分指定了 Service 所需的虚拟端口号为 8081，由于与 Pod 容器端口号 8080 不一样，所以需要再通过 targetPort 来指定后端 Pod 的端口号。selector 定义部分设置的是后端 Pod 所拥有的 label: app=webapp。

创建该 Service 并查看其 ClusterIP 地址：

```
# kubectl create -f webapp-svc.yaml
service "webapp" created
```

```
# kubectl get svc
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
webapp   169.169.28.190    <none>          8081/TCP    3s
```

通过 Service 的 IP 地址和 Service 的端口号进行访问：

```
# curl 169.169.28.190:8081
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
  ....
```

同样，对 Service 地址 169.169.28.190:8081 的访问被自动负载分发到了后端两个 Pod 之一：172.17.1.3:8080 或 172.17.1.4:8080。目前 Kubernetes 提供了两种负载分发策略：RoundRobin 和 SessionAffinity，具体说明如下。

- RoundRobin：轮询模式，即轮询将请求转发到后端的各个 Pod 上。
- SessionAffinity：基于客户端 IP 地址进行会话保持的模式，即第一次将某个客户端发起的请求转发到后端的某个 Pod 上，之后从相同的客户端发起的请求都将被转发到后端相同的 Pod 上。

在默认情况下，Kubernetes 采用 RoundRobin 模式进行路由选择，但我们也可以通过将 service.spec.sessionAffinity 设置为“ClientIP”来启用 SessionAffinity 策略，这样，同一个客户端发来的请求就会建立一个 Session，并且对应到后端固定的某个 Pod 上了。

在某些应用场景中，开发人员希望自己控制负载均衡的策略，不使用 Service 提供的默认负载均衡的功能，Kubernetes 通过 Headless Service 的概念来实现这种功能，即不给 Service 设置 ClusterIP（无入口 IP 地址），而仅通过 Label Selector 将后端的 Pod 列表返回给调用的客户端。例如：

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
  clusterIP: None
  selector:
    app: nginx
```

该 Service 没有虚拟的 ClusterIP 地址，对其进行访问将获得具有 Label “app=nginx”的全部 Pod 列表，然后客户端程序需要实现自己的负载分发策略，再确定访问具体哪一个后端的 Pod。

在某些环境中，应用系统需要将一个外部数据库作为后端服务进行连接，或将另一个集群或 Namespace 中的服务作为服务的后端，这时可以通过创建一个无 Label Selector 的 Service 来实现：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

通过该定义创建的是一个不带标签选择器的 Service，即无法选择后端的 Pod，系统不会自动创建 Endpoint，因此需要手动创建一个和该 Service 同名的 Endpoint，用于指向实际的后端访问地址。创建 Endpoint 的配置文件内容如下：

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
- addresses:
  - IP: 1.2.3.4
  ports:
  - port: 80
```

如图 2.16 所示，访问没有标签选择器的 Service 和带有标签选择器的 Service 一样，请求将会被路由到由用户手动定义的后端 Endpoint 上。

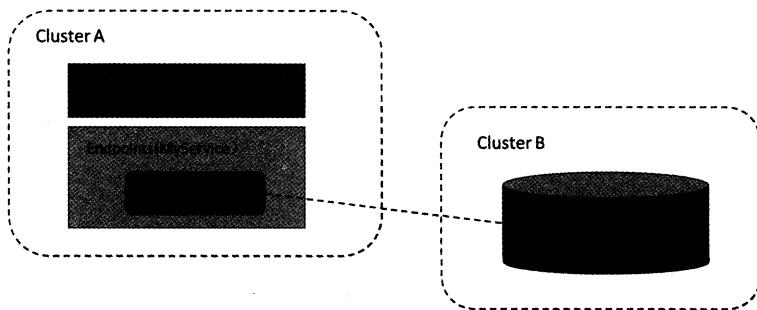


图 2.16 Service 指向外部服务

有时，一个容器应用也可能提供多个端口的服务，所以在 Service 的定义中也可以相应地设置为多个端口号。在下面的例子中，Service 设置了两个端口号，并且为每个端口号进行了命名：

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  ports:
    - port: 8080
      targetPort: 8080
      name: web
    - port: 8005
      targetPort: 8005
      name: management
  selector:
    app: webapp
```

另一个例子是两个端口号使用了不同的 4 层协议，即 TCP 或 UDP：

```
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 169.169.0.100
  ports:
    - name: dns
      port: 53
```

```
protocol: UDP
- name: dns-tcp
  port: 53
  protocol: TCP
```

2.5.3 集群外部访问 Pod 或 Service

由于 Pod 和 Service 是 Kubernetes 集群范围内的虚拟概念，所以集群外的客户端系统无法通过 Pod 的 IP 地址或者 Service 的虚拟 IP 地址和虚拟端口号访问到它们。为了让外部客户端可以访问这些服务，可以将 Pod 或 Service 的端口号映射到宿主机，以使得客户端应用能够通过物理机访问容器应用。

1. 将容器应用的端口号映射到物理机

(1) 通过设置容器级别的 hostPort，将容器应用的端口号映射到物理机上：

```
pod-hostport.yaml
apiVersion: v1
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  containers:
  - name: webapp
    image: tomcat
    ports:
      - containerPort: 8080
        hostPort: 8081
```

通过 kubectl create 命令创建这个 Pod：

```
# kubectl create -f pod-hostport.yaml
pod "webapp" created
```

通过物理机的 IP 地址和 8081 端口号访问 Pod 内的容器服务：

```
# curl 192.168.18.3:8081
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
  ....
```

(2) 通过设置 Pod 级别的 hostNetwork=true，该 Pod 中所有容器的端口号都将被直接映射到物理机上。设置 hostNetwork=true 时需要注意，在容器的 ports 定义部分如果不指定 hostPort，则默认 hostPort 等于 containerPort，如果指定了 hostPort，则 hostPort 必须等于 containerPort 的值。

```
pod-hostnetwork.yaml
apiVersion: v1
kind: Pod
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  hostNetwork: true
  containers:
    - name: webapp
      image: tomcat
      imagePullPolicy: Never
      ports:
        - containerPort: 8080
```

创建这个 Pod:

```
# kubectl create -f pod-hostnetwork.yaml
pod "webapp" created
```

通过物理机的 IP 地址和 8080 端口号访问 Pod 内的容器服务:

```
# curl 192.168.18.4:8080
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
  .....
```

2. 将 Service 的端口号映射到物理机

(1) 通过设置 nodePort 映射到物理机，同时设置 Service 的类型为 NodePort:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 8080
```

```
nodePort: 8081
selector:
  app: webapp
```

创建这个 Service:

```
# kubectl create -f webapp-svc-nodeport.yaml
You have exposed your service on an external port on all nodes in your
cluster. If you want to expose this service to the external internet, you may
need to set up firewall rules for the service port(s) (tcp:8081) to serve traffic.
```

See <http://releases.k8s.io/release-1.3/docs/user-guide/services-firewalls.md> for more details.

```
service "webapp" created
```

系统提示信息说明：由于要使用物理机的端口号，所以需要在防火墙上做好相应的配置，以使得外部客户端能够访问到该端口。

通过物理机的 IP 地址和 nodePort 8081 端口号访问服务：

```
# curl 192.168.18.3:8081
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Apache Tomcat/8.0.35</title>
  ....
```

同样，对该 Service 的访问也将被负载分发到后端的多个 Pod 上。

(2) 通过设置 LoadBalancer 映射到云服务商提供的 LoadBalancer 地址。这种用法仅用于在公有云服务提供商的云平台上设置 Service 的场景。在下面的例子中，status.loadBalancer.ingress.ip 设置的 146.148.47.155 为云服务商提供的负载均衡器的 IP 地址。对该 Service 的访问请求将会通过 LoadBalancer 转发到后端 Pod 上，负载分发的实现方式则依赖于云服务商提供的 LoadBalancer 的实现机制。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
    nodePort: 30061
```

```

clusterIP: 10.0.171.239
loadBalancerIP: 78.11.24.19
type: LoadBalancer
status:
  loadBalancer:
    ingress:
      - ip: 146.148.47.155

```

2.5.4 DNS 服务搭建指南

根据第1章对Service概念的说明,为了能够通过服务的名字在集群内部进行服务的相互访问,需要创建一个虚拟的DNS服务来完成服务名到ClusterIP的解析。本节将对如何搭建DNS服务进行详细说明。

Kubernetes提供的虚拟DNS服务名为skydns,由四个组件组成。

- (1) etcd: DNS存储。
- (2) kube2sky: 将Kubernetes Master中的Service(服务)注册到etcd。
- (3) skyDNS: 提供DNS域名解析服务。
- (4) healthz: 提供对skydns服务的健康检查功能。

图2.17描述了Kubernetes DNS服务的总体架构。

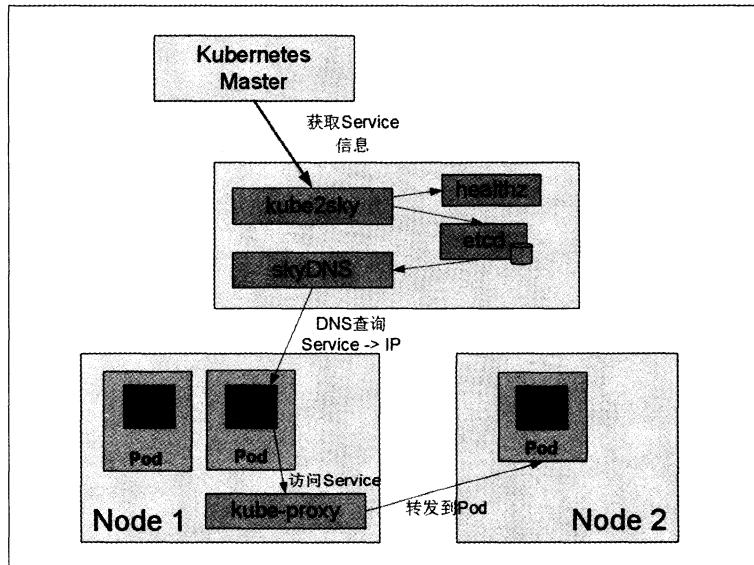


图2.17 Kubernetes DNS服务的总体架构

1. skydns 配置文件说明

skydns 服务由一个 RC 和一个 Service 的定义组成，分别由配置文件 skydns-rc.yaml 和 skydns-svc.yaml 定义。

skydns 的 RC 配置文件 skydns-rc.yaml 的内容如下，包含了 4 个容器的定义：

```
skydns-rc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-dns-v11
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    version: v11
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-dns
    version: v11
  template:
    metadata:
      labels:
        k8s-app: kube-dns
        version: v11
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
        - name: etcd
          image: gcr.io/google_containers/etcd-amd64:2.2.1
          resources:
            limits:
              cpu: 100m
              memory: 50Mi
            requests:
              cpu: 100m
              memory: 50Mi
          command:
            - /usr/local/bin/etcd
            - --data-dir
            - /tmp/data
            - --listen-client-urls
            - http://127.0.0.1:2379,http://127.0.0.1:4001
            - --advertise-client-urls
            - http://127.0.0.1:2379,http://127.0.0.1:4001
```

```
- --initial-cluster-token
- skydns-etcd
volumeMounts:
- name: etcd-storage
  mountPath: /tmp/data
- name: kube2sky
image: gcr.io/google_containers/kube2sky-amd64:1.15
resources:
limits:
cpu: 100m
# Kube2sky watches all pods.
memory: 50Mi
requests:
cpu: 100m
memory: 50Mi
livenessProbe:
httpGet:
path: /healthz
port: 8080
scheme: HTTP
initialDelaySeconds: 60
timeoutSeconds: 5
successThreshold: 1
failureThreshold: 5
readinessProbe:
httpGet:
path: /readiness
port: 8081
scheme: HTTP
# we poll on pod startup for the Kubernetes master service and
# only setup the /readiness HTTP server once that's available.
initialDelaySeconds: 30
timeoutSeconds: 5
args:
# command = "/kube2sky"
- --kube-master-url=http://192.168.18.3:8080
- --domain=cluster.local
- name: skydns
image: gcr.io/google_containers/skydns:2015-10-13-8c72f8c
resources:
limits:
cpu: 100m
memory: 50Mi
requests:
cpu: 100m
memory: 50Mi
args:
```

```

# command = "/skydns"
- --machines=http://127.0.0.1:4001
- --addr=0.0.0.0:53
- --ns-rotate=false
- --domain=cluster.local
ports:
- containerPort: 53
  name: dns
  protocol: UDP
- containerPort: 53
  name: dns-tcp
  protocol: TCP
- name: healthz
image: gcr.io/google_containers/exechealthz:1.0
resources:
  # keep request = limit to keep this container in guaranteed class
limits:
  cpu: 10m
  memory: 20Mi
requests:
  cpu: 10m
  memory: 20Mi
args:
- --cmd=nslookup kubernetes.default.svc.cluster.local 127.0.0.1 >/dev/null
- --port=8080
ports:
- containerPort: 8080
  protocol: TCP
volumes:
- name: etcd-storage
  emptyDir: {}
dnsPolicy: Default # Don't use cluster DNS.

```

需要修改的几个配置参数如下。

(1) kube2sky 容器需要访问 Kubernetes Master，需要配置 Master 所在物理主机的 IP 地址和端口号，本例中设置参数--kube_master_url 的值为 http://192.168.18.3:8080。

(2) kube2sky 容器和 skydns 容器的启动参数--domain，设置 Kubernetes 集群中 Service 所属的域名，本例中为“cluster.local”。启动后，kube2sky 会通过 API Server 监控集群中全部 Service 的定义，生成相应的记录并保存到 etcd 中。kube2sky 为每个 Service 生成以下两条记录。

- ◎ <service_name>.<namespace_name>.<domain>。
- ◎ <service_name>.<namespace_name>.svc.<domain>。

(3) skydns 的启动参数--addr=0.0.0.0:53 表示使用本机 TCP 和 UDP 的 53 端口提供服务。

skydns 的 Service 配置文件 skydns-svc.yaml 的内容如下：

```
skydns-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeDNS"
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 169.169.0.100
  ports:
    - name: dns
      port: 53
      protocol: UDP
    - name: dns-tcp
      port: 53
      protocol: TCP
```

注意，skydns 服务使用的 clusterIP 需要我们指定一个固定的 IP 地址，每个 Node 的 kubelet 进程都将使用这个 IP 地址，不能通过 Kubernetes 自动分配。

另外，这个 IP 地址需要在 kube-apiserver 启动参数--service-cluster-ip-range 指定的 IP 地址范围内。

在创建 skydns 容器之前，先修改每个 Node 上 kubelet 的启动参数。

2. 修改每台 Node 上的 kubelet 启动参数

修改每台 Node 上 kubelet 的启动参数，加上以下两个参数。

- ◎ --cluster_dns=169.169.0.100：为 DNS 服务的 ClusterIP 地址。
- ◎ --cluster_domain=cluster.local：为 DNS 服务中设置的域名。

然后重启 kubelet 服务。

3. 创建 skydns RC 和 Service

通过 kubectl create 完成 skydns 的 RC 和 Service 的创建：

```
# kubectl create -f skydns-rc.yaml
```

```
# kubectl create -f skydns-svc.yaml
```

查看 RC、Pod 和 Service，确保容器成功启动：

```
# kubectl get rc --namespace=kube-system
```

NAME	DESIRED	CURRENT	AGE
kube-dns-v11	1	1	1d

```
# kubectl get pods --namespace=kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
kube-dns-v11-6dlwu	4/4	Running	0	1d

```
# kubectl get services --namespace=kube-system
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kube-dns	169.169.0.100	<none>	53/UDP, 53/TCP	1d

然后，我们为 redis-master 应用创建一个 Service：

redis-master-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    name: redis-master
```

查看创建好的 redis-master service：

```
# kubectl get services
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
redis-master	169.169.8.10	<none>	6379/TCP	1h

可以看到，系统为 redis-master 服务分配了一个虚拟 IP 地址：169.169.8.10。

到此，在 Kubernetes 集群内的虚拟 DNS 服务就搭建好了。在需要访问 redis-master 的应用中，仅需要配置上 redis-master Service 的名称和服务的端口号，就能够访问到 redis-master 应用了，让我们回顾一下 redis-slave 应用需要访问 redis-master 的配置内容：

redis-slave 镜像的启动脚本/run.sh 的内容：

```
if [[ ${GET_HOSTS_FROM:-dns} == "env" ]]; then
  redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST} 6379
else
  redis-server --slaveof redis-master 6379
```

```
fi
```

在使用 DNS 模式的情况下，redis-slave 配置的 Master 地址为：redis-master:6379。通过服务名进行配置，能够极大地简化客户端应用对后端服务变化的感知，包括服务虚拟 IP 地址的变化、服务后端 Pod 的变化等，对应用程序的微服务架构实现提供了强有力的支撑。

4. 通过 DNS 查找 Service

接下来使用一个带有 nslookup 工具的 Pod 来验证 DNS 服务是否能够正常工作：

busybox.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - name: busybox
    image: gcr.io/google_containers/busybox
    command:
      - sleep
      - "3600"
```

运行 `kubectl create -f busybox.yaml` 完成创建。

在该容器成功启动后，通过 `kubectl exec <container_id> nslookup` 进行测试：

```
# kubectl exec busybox -- nslookup redis-master
Server: 169.169.0.100
Address 1: 169.169.0.100

Name:      redis-master
Address 1: 169.169.8.10
```

可以看到，通过 DNS 服务器 169.169.0.100 成功找到了名为“redis-master”服务的 IP 地址：169.169.8.10。

如果某个 Service 属于不同的命名空间，那么在进行 Service 查找时，需要带上 `namespace` 的名字。下面以查找 kube-dns 服务为例：

```
# kubectl exec busybox -- nslookup kube-dns.kube-system
Server: 169.169.0.100
Address 1: 169.169.0.100

Name:      kube-dns.kube-system
Address 1: 169.169.0.100
```

如果仅使用“kube-dns”来进行查找，则将会失败：

```
nslookup: can't resolve 'kube-dns'
```

5. DNS 服务的工作原理解析

让我们看看 DNS 服务背后的工作原理。

(1) kube2sky 容器应用通过调用 Kubernetes Master 的 API 获得集群中所有 Service 的信息，并持续监控新 Service 的生成，然后写入 etcd 中。

查看 etcd 中存储的 Service 信息：

```
# kubectl exec kube-dns-v8-5tpm2 -c etcd --namespace=kube-system etcdctl ls  
/skydns/local/cluster  
/skydns/local/cluster/default  
/skydns/local/cluster/svc  
/skydns/local/cluster/kube-system
```

可以看到在 skydns 键下面，根据我们配置的域名（cluster.local）生成了 local/cluster 子键，接下来是 namespace（default 和 kube-system）和 svc（下面也按 namespace 生成子键）。

查看 redis-master 服务对应的键值：

```
# kubectl exec kube-dns-v8-5tpm2 -c etcd --namespace=kube-system etcdctl get /  
skydns/local/cluster/default/redis-master  
{"host": "169.169.8.10", "priority": 10, "weight": 10, "ttl": 30, "targetstrip": 0}
```

可以看到，redis-master 服务对应的完整域名为 redis-master.default.cluster.local，并且其 IP 地址为 169.169.8.10。

(2) 根据 kubelet 启动参数的设置（--cluster_dns），kubelet 会在每个新创建的 Pod 中设置 DNS 域名解析配置文件/etc/resolv.conf 文件，在其中增加了一条 nameserver 配置和一条 search 配置：

```
nameserver 169.169.0.100  
search default.svc.cluster.local svc.cluster.local cluster.local localdomain  
通过名字服务器 169.169.0.100 访问的实际上就是 skydns 在 53 端口上提供的 DNS 解析服务。
```

(3) 最后，应用程序就能够像访问网站域名一样，仅仅通过服务的名字就能访问到服务了。

仍然以 redis-slave 为例，假设已经启动了 redis-slave Pod，登录 redis-slave 容器进行查看，可以看到其通过 DNS 域名服务找到了 redis-master 的 IP 地址 169.169.8.10，并成功建立了连接。

2.5.5 Ingress: HTTP 7层路由机制

根据前面对 Service 的使用说明，我们知道 Service 的表现形式为 IP:Port，即工作在 TCP/IP 层。而对于基于 HTTP 的服务来说，不同的 URL 地址经常对应到不同的后端服务或者虚拟服务器（Virtual Host），这些应用层的转发机制仅通过 Kubernetes 的 Service 机制是无法实现的。Kubernetes v1.1 版本中新增的 Ingress 将不同 URL 的访问请求转发到后端不同的 Service，实现 HTTP 层的业务路由机制。在 Kubernetes 集群中，Ingress 的实现需要通过 Ingress 的定义与 Ingress Controller 的定义结合起来，才能形成完整的 HTTP 负载分发功能。

图 2.18 显示了一个典型 HTTP 层路由的例子。

- ◎ 对 `http://mywebsite.com/api` 的访问将被路由到后端名为“api”的 Service。
- ◎ 对 `http://mywebsite.com/web` 的访问将被路由到后端名为“web”的 Service。
- ◎ 对 `http://mywebsite.com/doc` 的访问将被路由到后端名为“doc”的 Service。

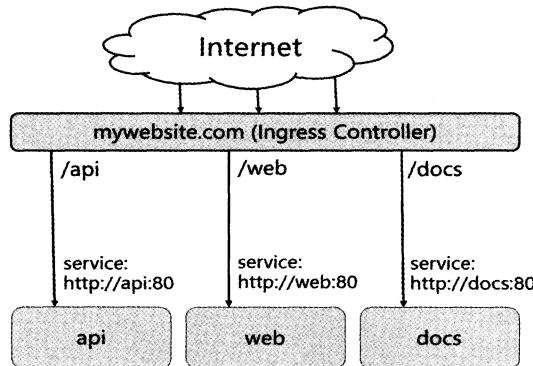


图 2.18 Ingress 示例

1. 创建 Ingress Controller

在定义 Ingress 之前，需要先部署 Ingress Controller，以实现为所有后端 Service 提供一个统一的入口。Ingress Controller 需要实现基于不同 HTTP URL 向后转发的负载分发规则，通常应该根据应用系统的需求进行个性化实现。如果公有云服务商能够提供该类型的 HTTP 路由 LoadBalancer，则也可设置其为 Ingress Controller。

在 Kubernetes 中，Ingress Controller 将以 Pod 的形式运行，监控 apiserver 的/ingress 接口（在 1.3 版本中为`/apis/extensions/v1beta1/namespaces/<namespace_name>/ingresses`接口）后端的 backend services，如果 service 发生变化，则 Ingress Controller 应自动更新其转发规则。

在下面的例子中，我们使用 Nginx 来实现一个 Ingress Controller，需要实现的基本逻辑如下。

- (1) 监听 apiserver，获取全部 ingress 的定义。
- (2) 基于 ingress 的定义，生成 Nginx 所需的配置文件/etc/nginx/nginx.conf。
- (3) 执行 nginx -s reload 命令，重新加载 nginx.conf 配置文件的内容，

基于 Go 语言的代码实现如下：

```
for {
    rateLimiter.Accept()
    ingresses, err := ingClient.List(labels.Everything(), fields.Everything())
    if err != nil || reflect.DeepEqual(ingresses.Items, known.Items) {
        continue
    }
    if w, err := os.Create("/etc/nginx/nginx.conf"); err != nil {
        log.Fatalf("Failed to open %v: %v", nginxConf, err)
    } else if err := tmpl.Execute(w, ingresses); err != nil {
        log.Fatalf("Failed to write template %v", err)
    }
    shellOut("nginx -s reload")
}
```

我们可以通过直接下载谷歌提供的 nginx-ingress 镜像来创建 Ingress Controller：

```
nginx-ingress-rc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-ingress
  labels:
    app: nginx-ingress
spec:
  replicas: 1
  selector:
    app: nginx-ingress
  template:
    metadata:
      labels:
        app: nginx-ingress
    spec:
      containers:
        - image: gcr.io/google_containers/nginx-ingress:0.1
          name: nginx
          ports:
            - containerPort: 80
              hostPort: 80
```

这里，该 Nginx 应用设置了 hostPort，即它将容器应用监听的 80 端口号映射到物理机，以使得客户端应用可以通过 URL 地址“<http://物理机 IP:80>”来访问该 Ingress Controller。

通过 kubectl create 命令创建该 RC:

```
# kubectl create -f nginx-ingress-rc.yaml
replicationcontroller "nginx-ingress" created
```

```
# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
nginx-ingress-mrwzt  1/1     Running   0          2s
```

2. 定义 Ingress

为 mywebsite.com 定义 Ingress，设置到后端 Service 的转发规则:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: mywebsite-ingress
spec:
  rules:
  - host: mywebsite.com
    http:
      paths:
      - path: /web
        backend:
          serviceName: webapp
          servicePort: 80
```

这个 Ingress 的定义说明对目标 URL `http://mywebsite.com/web` 的访问将被转发到 Kubernetes 的一个 Service 上: `webapp:80`。

创建该 Ingress:

```
# kubectl create -f ingress.yaml
ingress "mywebsite-ingress" created
```

```
# kubectl get ingress
NAME           HOSTS             ADDRESS   PORTS   AGE
mywebsite-ingress  mywebsite.com        80       17s
```

在该 Ingress 成功创建后，登录 `nginx-ingress` Pod，查看其自动生成的 `nginx.conf` 配置文件内容:

```
events {
  worker_connections 1024;
}
http {
  server {
    listen 80;
    server_name mywebsite.com;  # Ingress 中定义的虚拟 host 名
```

```
resolver 127.0.0.1;

location /web {          # Ingress 中定义的路径 /web
    proxy_pass http://webapp; # service 名
}
}
```

3. 访问 <http://mywebsite.com/web>

由于 Ingress Controller 设置了 hostPort，所以我们可以通过其所在的物理机对其进行访问。可以在物理机上设置 mywebsite.com 对应的 IP 地址，也可以通过 curl --resolve 进行指定：

```
$ curl --resolve mywebsite.com:80:192.168.18.3 mywebsite.com/foo
```

将获得 Kubernetes Service “webapp:80” 提供的主页。

4. Ingress 的发展路线

当前的 Ingress 还是 beta 版本，在 Kubernetes 的后续版本中将增加至少以下功能。

- ◎ 支持更多 TLS 选项，例如 SNI、重加密等。
- ◎ 支持 L4 和 L7 负载均衡策略（目前只支持 HTTP 层的规则）。
- ◎ 支持更多的转发规则（目前仅支持基于 URL 路径的），例如重定向规则、会话保持规则等。

第3章

Kubernetes 核心原理

本章对 Kubernetes 的核心原理进行深入分析，首先从 API Server 的访问开始讲起，然后分析 Master 节点上 Controller Manager 各个组件的功能实现，以及 Scheduler 预选算法和优选算法。接下来讲解 Node 节点上的 kubelet 和 kube-proxy 组件的运行机制。最后，深入分析安全机制和网络原理。

3.1 Kubernetes API Server 原理分析

总体来看，Kubernetes API Server 的核心功能是提供了 Kubernetes 各类资源对象（如 Pod、RC、Service 等）的增、删、改、查及 Watch 等 HTTP Rest 接口，成为集群内各个功能模块之间数据交互和通信的中心枢纽，是整个系统的数据总线和数据中心。除此之外，它还有以下一些功能特性。

- (1) 是集群管理的 API 入口。
- (2) 是资源配置控制的入口。
- (3) 提供了完备的集群安全机制。

3.1.1 Kubernetes API Server 概述

Kubernetes API Server 通过一个名为 kube-apiserver 的进程提供服务，该进程运行在 Master 节点上。在默认情况下，kube-apiserver 进程在本机的 8080 端口（对应参数--insecure-port）提供 REST 服务。我们可以同时启动 HTTPS 安全端口（--secure-port=6443）来启动安全机制，加

强 REST API 访问的安全性。

通常我们可以通过命令行工具 kubectl 来与 Kubernetes API Server 交互，它们之间的接口是 REST 调用。为了测试和学习 Kubernetes API Server 所提供的接口，我们也可以使用 curl 命令行工具进行快速验证。

比如，我们登录 Master 节点，运行下面的 curl 命令，得到以 JSON 方式返回的 Kubernetes API 的版本信息：

```
# curl localhost:8080/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "192.168.18.131:6443"
    }
  ]
}
```

可以运行下面的命令，来查看 Kubernetes API Server 目前所支持的资源对象的种类：

```
# curl localhost:8080/api/v1
```

根据以上命令的输出，我们可以运行下面的 curl 命令，分别返回集群中的 Pod 列表、Service 列表、RC 列表等：

```
# curl localhost:8080/api/v1/pods
# curl localhost:8080/api/v1/services
# curl localhost:8080/api/v1/replicationcontrollers
```

如果我们只想对外暴露部分 REST 服务，则可以在 Master 或其他任何节点上通过运行 kubectl proxy 进程启动一个内部代理来实现。

运行下面的命令，我们在 8001 端口启动代理，并且拒绝客户端访问 RC 的 API：

```
# kubectl proxy --reject-paths="/api/v1/replicationcontrollers" --port=8001
--v=2
Starting to serve on 127.0.0.1:8001
```

运行下面的命令进行验证：

```
# curl localhost:8001/api/v1/replicationcontrollers
<h3>Unauthorized</h3>
```

kubectl proxy 具有很多特性，最实用的一个特性是提供简单有效的安全机制，比如采用白名单来限制非法客户端访问时，只要增加下面这个参数即可：

```
--accept-hosts="^localhost$, ^127\\.0\\.0\\.1$, ^\\[::1\\]$"
```

最后一种方式是通过编程的方式调用 Kubernetes API Server。具体使用场景又细分为以下两种。

第1种使用场景：运行在Pod里的用户进程调用Kubernetes API，通常用来实现分布式集群搭建的目标。比如下面这段来自谷歌官方的Elastic Search集群例子中的代码，Pod在启动的过程中通过访问Endpoints的API，找到属于elasticsearch-logging这个Service的所有Pod副本的IP地址，用来构建集群，如图3.1所示。

```

if elasticsearch == nil {
    glog.Warningf("Failed to find the elasticsearch-logging service: %v", err)
    return
}

var endpoints *api.Endpoints
addrs := []string{}
// Wait for some endpoints.
count := 0
for t := time.Now(); time.Since(t) < 5*time.Minute; time.Sleep(10 * time.Second) {
    endpoints, err = c.Endpoints(api.NamespaceSystem).Get("elasticsearch-logging")
    if err != nil {
        continue
    }
    addrs = flattenSubsets(endpoints.Subsets)
    glog.Infof("Found %s", addrs)
    if len(addrs) > 0 && len(addrs) == count {
        break
    }
    count = len(addrs)
}

glog.Infof("Endpoints = %s", addrs)
fmt.Printf("discovery.zen.ping.unicast.hosts: [%s]\n", strings.Join(addrs, ","))
export NODE_MASTER=${NODE_MASTER:-true}
export NODE_DATA=${NODE_DATA:-true}
/elasticsearch_logging_discovery >> /elasticsearch-1.5.2/config/elasticsearch.yml
export HTTP_PORT=${HTTP_PORT:-9200}
export TRANSPORT_PORT=${TRANSPORT_PORT:-9300}
/elasticsearch-1.5.2/bin/elasticsearch

```

图3.1 应用程序编程访问API Server

在上述使用场景中，Pod中的进程如何知道API Server的访问地址呢？答案很简单，因为Kubernetes API Server本身也是一个Service，它的名字就是“kubernetes”，并且它的Cluster IP地址是Cluster IP地址池里的第1个地址！另外，它所服务的端口是HTTPS端口443，通过kubectl get service命令可以确认这一点：

```
# kubectl get service
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes   169.169.0.1    <none>           443/TCP     30d
```

第2种使用场景：开发基于Kubernetes的管理平台。比如调用Kubernetes API来完成Pod、Service、RC等资源对象的图形化创建和管理界面，此时可以使用Kubernetes及各开源社区为开发人员提供的各种语言版本的Client Library。我们会在后面介绍通过编程方式访问API Server的一些细节技术。

3.1.2 独特的 Kubernetes Proxy API 接口

前面我们说过，Kubernetes API Server 最主要的 REST 接口是资源对象的增、删、改、查，除此之外，它还提供了一类很特殊的 REST 接口——Kubernetes Proxy API 接口，这类接口的作用是代理 REST 请求，即 Kubernetes API Server 把收到的 REST 请求转发到某个 Node 上的 kubelet 守护进程的 REST 端口上，由该 Kubelet 进程负责响应。

首先，我们来说说 Kubernetes Proxy API 里关于 Node 的相关接口，该接口的 REST 路径为 /api/v1/proxy/nodes/{name}，其中 {name} 为节点的名称或 IP 地址，包括以下几个具体接口：

- ◎ /api/v1/proxy/nodes/{name}/pods/ #列出指定节点内所有 Pod 的信息
- ◎ /api/v1/proxy/nodes/{name}/stats/ #列出指定节点内物理资源的统计信息
- ◎ /api/v1/proxy/nodes/{name}/spec/ #列出指定节点的概要信息

例如当前 Node 节点的名字为 k8s-node-1，用下面的命令即可获取该节点上所有运行中的 Pod：

```
# curl localhost:8080/api/v1/proxy/nodes/k8s-node-1/pods
```

需要说明的是：这里获取的 Pod 的信息数据来自 Node 而非 etcd 数据库，所以两者可能在某些时间点会有偏差。此外，如果 kubelet 进程在启动时包含--enable-debugging-handlers=true 参数，那么 Kubernetes Proxy API 还会增加下面的接口：

- ◎ /api/v1/proxy/nodes/{name}/run #在节点上运行某个容器
- ◎ /api/v1/proxy/nodes/{name}/exec #在节点上的某个容器中运行某条命令
- ◎ /api/v1/proxy/nodes/{name}/attach #在节点上 attach 某个容器
- ◎ /api/v1/proxy/nodes/{name}/portForward #实现节点上的 Pod 端口转发
- ◎ /api/v1/proxy/nodes/{name}/logs #列出节点的各类日志信息，例如 tallylog、lastlog、wtmp、ppp/、rhsml/、audit/、tuned/ 和 anaconda/ 等
- ◎ /api/v1/proxy/nodes/{name}/metrics #列出和该节点相关的 Metrics 信息
- ◎ /api/v1/proxy/nodes/{name}/runningpods #列出节点内运行中的 Pod 信息
- ◎ /api/v1/proxy/nodes/{name}/debug/pprof #列出节点内当前 Web 服务的状态，包括 CPU 占用情况和内存使用情况等

接下来，我们来说说 Kubernetes Proxy API 里关于 Pod 的相关接口，通过这些接口，我们可以访问 Pod 里某个容器提供的服务（如 Tomcat 在 8080 端口服务）：

- ◎ /api/v1/proxy/namespaces/{namespace}/pods/{name}/{path: *} #访问 Pod 的某个服务接口
- ◎ /api/v1/proxy/namespaces/{namespace}/pods/{name} #访问 Pod
- ◎ /api/v1/namespaces/{namespace}/pods/{name}/proxy/{path: *} #访问 Pod 的某个服务接口
- ◎ /api/v1/namespaces/{namespace}/pods/{name}/proxy #访问 Pod

在上面的 4 个接口里，后面两个接口的功能与前面两个完全一样，只是写法不同。下面我们用第 1 章的 Java Web 例子中的 Tomcat Pod 来说明上述 Proxy 接口的用法。

首先，得到 Pod 的名字：

```
# kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
mysql-c95jc   1/1     Running   0          8d
myweb-g9pmm   1/1     Running   0          8d
```

然后，运行下面的命令，会输出 Tomcat 的首页，即相当于访问 <http://localhost:8080/>：

```
# curl http://localhost:8080/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/
```

我们也可以在浏览器中访问上面的地址，比如 Master 节点的 IP 地址是 192.168.18.131，我们在浏览器中输入 <http://192.168.18.131:8080/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/>，就能够访问 Tomcat 首页了；而如果输入 [/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/demo](http://192.168.18.131:8080/api/v1/proxy/namespaces/default/pods/myweb-g9pmm/demo)，就能访问 Tomcat 中 Demo 应用的页面了。

看到这里，你可能明白 Pod 的 Proxy 接口的作用和意义了：在 Kubernetes 集群之外访问某个 Pod 容器的服务（HTTP 服务）时，可以用 Proxy API 实现，这种场景多用于管理目的，比如逐一排查 Service 的 Pod 副本，检查哪些 Pod 的服务存在异常问题。

最后我们说说 Service，Kubernetes Proxy API 也有 Service 的 Proxy 接口，其接口定义与 Pod 的接口定义基本一样：`/api/v1/proxy/namespaces/{namespace}/services/{name}`。比如，我们想访问 myweb 这个 Service，则可以在浏览器里输入 [http://192.168.18.131:8080/api/v1/proxy/namespaces/default/services/myweb/demo/](http://192.168.18.131:8080/api/v1/proxy/namespaces/default/services/myweb/demo)。

3.1.3 集群功能模块之间的通信

从图 3.2 中可以看出，Kubernetes API Server 作为集群的核心，负责集群各功能模块之间的通信。集群内的各个功能模块通过 API Server 将信息存入 etcd，当需要获取和操作这些数据时，则通过 API Server 提供的 REST 接口（用 GET、LIST 或 WATCH 方法）来实现，从而实现各模块之间的信息交互。

常见的一个交互场景是 kubelet 进程与 API Server 的交互。每个 Node 节点上的 kubelet 每隔一个时间周期，就会调用一次 API Server 的 REST 接口报告自身状态，API Server 收到这些信息后，将节点状态信息更新到 etcd 中。此外，kubelet 也通过 API Server 的 Watch 接口监听 Pod 信息，如果监听到新的 Pod 副本被调度绑定到本节点，则执行 Pod 对应的容器的创建和启动逻辑；如果监听到 Pod 对象被删除，则删除本节点上的相应的 Pod 容器；如果监听到修改 Pod 信息，则 kubelet 监听到变化后，会相应地修改本节点的 Pod 容器。

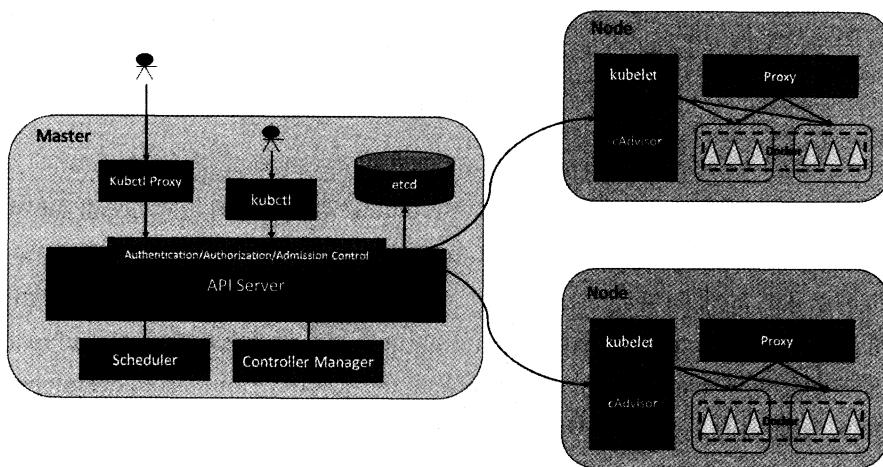


图 3.2 Kubernetes 结构图

另外一个交互场景是 `kube-controller-manager` 进程与 API Server 的交互。`kube-controller-manager` 中的 Node Controller 模块通过 API Server 提供的 Watch 接口，实时监控 Node 的信息，并做相应处理。

还有一个比较重要的交互场景是 `kube-scheduler` 与 API Server 的交互。当 Scheduler 通过 API Server 的 Watch 接口监听到新建 Pod 副本的信息后，它会检索所有符合该 Pod 要求的 Node 列表，开始执行 Pod 调度逻辑，调度成功后将 Pod 绑定到目标节点上。

为了缓解集群各模块对 API Server 的访问压力，各功能模块都采用缓存机制来缓存数据。各功能模块定时从 API Server 获取指定的资源对象信息（通过 LIST 及 WATCH 方法），然后将这些信息保存到本地缓存，功能模块在某些情况下不直接访问 API Server，而是通过访问缓存数据来间接访问 API Server。

3.2 Controller Manager 原理分析

Controller Manager 作为集群内部的管理控制中心，负责集群内的 Node、Pod 副本、服务端点（Endpoint）、命名空间（Namespace）、服务账号（ServiceAccount）、资源定额（ResourceQuota）等的管理，当某个 Node 意外宕机时，Controller Manager 会及时发现此故障并执行自动化修复流程，确保集群始终处于预期的工作状态。

如图 3.3 所示，Controller Manager 内部包含 Replication Controller、Node Controller、ResourceQuota Controller、Namespace Controller、ServiceAccount Controller、Token Controller、

Service Controller 及 Endpoint Controller 等多个 Controller，每种 Controller 都负责一种具体的控制流程，而 Controller Manager 正是这些 Controller 的核心管理者。

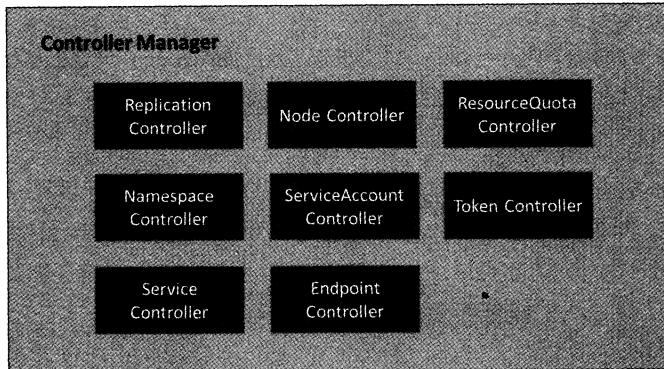


图 3.3 Controller Manager 结构图

一般来说，智能系统和自动系统通常会通过一个被称为“操纵系统”的机构来不断修正系统的工作状态。在 Kubernetes 集群中，每个 Controller 都是这样一个“操纵系统”，它们通过 API Server 提供的接口实时监控整个集群里的每个资源对象的当前状态，当发生各种故障导致系统状态发生变化时，会尝试着将系统状态从“现有状态”修正到“期望状态”。本节将详细分析 Controller Manager 的这些 Controller 的原理。

由于 ServiceAccount Controller 与 Token Controller 是与安全相关的两个控制器，并且与 Service Account、Token 密切相关，所以我们将对它们的分析放到后面的深入集群安全的章节中讲解。

在 Kubernetes 集群中与 Controller Manager 并重的另一个组件是 Kubernetes Scheduler，它的作用是将待调度的 Pod（包括通过 API Server 新创建的 Pod 及 RC 为补足副本而创建的 Pod 等）通过一些复杂的调度流程计算出最佳目标节点，然后绑定到该节点上。本章最后会介绍 Kubernetes Scheduler 调度器的基本原理。

3.2.1 Replication Controller

为了区分 Controller Manager 中的 Replication Controller（副本控制器）和资源对象 Replication Controller，我们将资源对象 Replication Controller 简写为 RC，而本节中的 Replication Controller 是指“副本控制器”，以便于后续分析。

Replication Controller 的核心作用是确保在任何时候集群中一个 RC 所关联的 Pod 副本数量保持预设值。如果发现 Pod 副本数量超过预期值，则 Replication Controller 会销毁一些 Pod 副本；

反之，Replication Controller 会自动创建新的 Pod 副本，直到符合条件的 Pod 副本数量达到预设值。需要注意的一点是：只有当 Pod 的重启策略是 Always 的时候（`RestartPolicy=Always`），Replication Controller 才会管理该 Pod 的操作（例如创建、销毁、重启等）。在通常情况下，Pod 对象被成功创建后不会消失，唯一的例外是当 Pod 处于 `succeeded` 或 `failed` 状态的时间过长（超时参数由系统设定）时，该 Pod 会被系统自动回收，管理该 Pod 的副本控制器将在其他工作节点上重新创建、运行该 Pod 副本。

RC 中的 Pod 模板就像一个模具，模具制作出来的东西一旦离开模具，它们之间就再也没有关系了。同样，一旦 Pod 被创建完毕，无论模板如何变化，甚至换成一个新的模板，也不会影响到已经创建的 Pod。此外，Pod 可以通过修改它的标签来实现脱离 RC 的管控。该方法可以用于将 Pod 从集群中迁移、数据修复等调试。对于被迁移的 Pod 副本，RC 会自动创建一个新的副本替换被迁移的副本。需要注意的是，删除一个 RC 不会影响它所创建的 Pod。如果想删除一个 RC 所控制的 Pod，则需要将该 RC 的副本数（`Replicas`）属性设置为 0，这样所有的 Pod 副本都会被自动删除。

我们最好不要越过 RC 而直接创建 Pod，因为 Replication Controller 会通过 RC 管理 Pod 副本，实现自动创建、补足、替换、删除 Pod 副本，这样能提高系统的容灾能力，减少由于节点崩溃等意外状况造成的损失。即使你的应用程序只用到一个 Pod 副本，我们也强烈建议使用 RC 来定义 Pod。

我们总结一下 Replication Controller 的职责，如下所述。

- (1) 确保当前集群中有且仅有 N 个 Pod 实例， N 是 RC 中定义的 Pod 副本数量。
- (2) 通过调整 RC 的 `spec.replicas` 属性值来实现系统扩容或者缩容。
- (3) 通过改变 RC 中的 Pod 模板（主要是镜像版本）来实现系统的滚动升级。

最后，我们总结一下 Replication Controller 的典型使用场景，如下所述。

- (1) 重新调度（Rescheduling）。如前面所提及的，不管你想运行 1 个副本还是 1000 个副本，副本控制器都能确保指定数量的副本存在于集群中，即使发生节点故障或 Pod 副本被终止运行等意外状况。
- (2) 弹性伸缩（Scaling）。手动或者通过自动扩容代理修改副本控制器的 `spec.replicas` 属性值，非常容易实现扩大或缩小副本的数量。
- (3) 滚动更新（Rolling Updates）。副本控制器被设计成通过逐个替换 Pod 的方式来辅助服务的滚动更新。推荐的方式是创建一个新的只有一个副本的 RC，若新的 RC 副本数量加 1，则旧的 RC 的副本数量减 1，直到这个旧的 RC 的副本数量为零，然后删除该旧的 RC。通过上述模式，即使在滚动更新的过程中发生了不可预料的错误，Pod 集合的更新也都在可控范围内。

在理想情况下，滚动更新控制器需要将准备就绪的应用考虑在内，并保证在集群中任何时刻都有足够数量的可用 Pod。

3.2.2 Node Controller

kubelet 进程在启动时通过 API Server 注册自身的节点信息，并定时向 API Server 汇报状态信息，API Server 接收到这些信息后，将这些信息更新到 etcd 中，etcd 中存储的节点信息包括节点健康状况、节点资源、节点名称、节点地址信息、操作系统版本、Docker 版本、kubelet 版本等。节点健康状况包含“就绪”(True)“未就绪”(False)和“未知”(Unknown)三种。

Node Controller 通过 API Server 实时获取 Node 的相关信息，实现管理和监控集群中的各个 Node 节点的相关控制功能，Node Controller 的核心工作流程如图 3.4 所示。

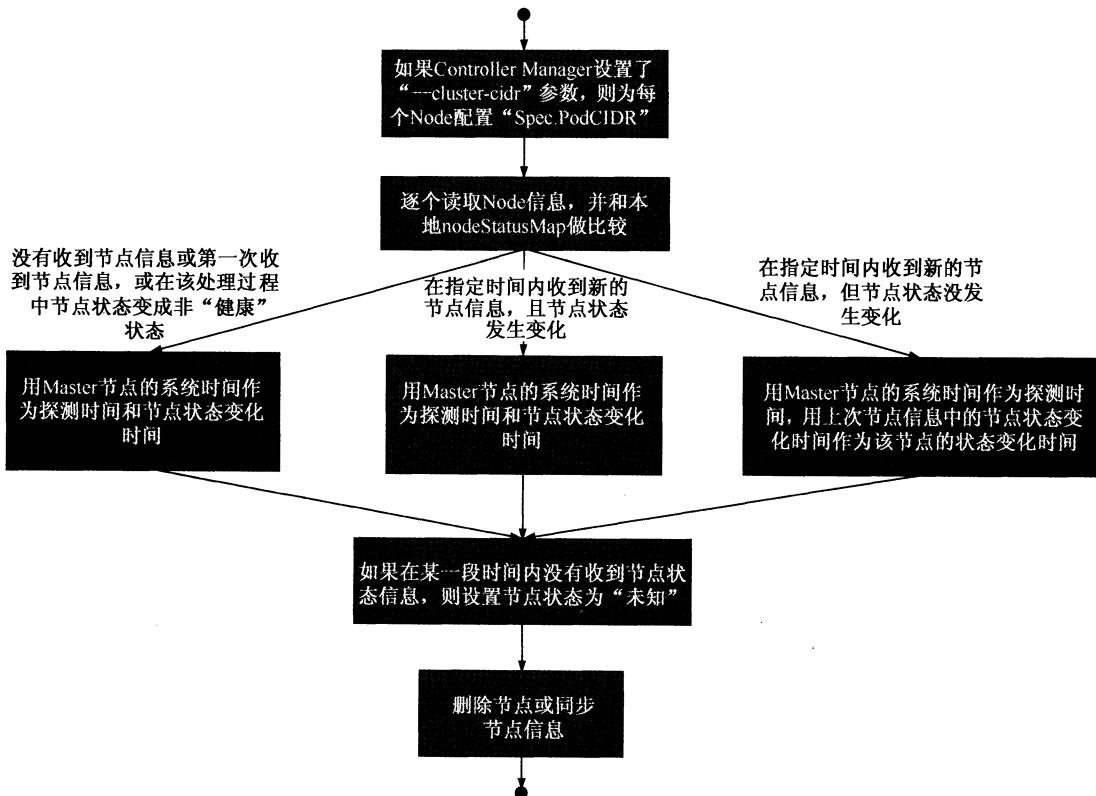


图 3.4 Node Controller 流程图

对流程中关键点的解释如下。

(1) Controller Manager 在启动时如果设置了`--cluster-cidr` 参数，那么为每个没有设置 Spec.PodCIDR 的 Node 节点生成一个 CIDR 地址，并用该 CIDR 地址设置节点的 Spec.PodCIDR 属性，这样做的目的是防止不同节点的 CIDR 地址发生冲突。

(2) 逐个读取节点信息，多次尝试修改 nodeStatusMap 中的节点状态信息，将该节点信息和 Node Controller 的 nodeStatusMap 中保存的节点信息做比较。如果判断出没有收到 kubelet 发送的节点信息、第 1 次收到节点 kubelet 发送的节点信息，或在该处理过程中节点状态变成非“健康”状态，则在 nodeStatusMap 中保存该节点的状态信息，并用 Node Controller 所在节点的系统时间作为探测时间和节点状态变化时间。如果判断出在指定时间内收到新的节点信息，且节点状态发生变化，则在 nodeStatusMap 中保存该节点的状态信息，并用 Node Controller 所在节点的系统时间作为探测时间和节点状态变化时间。如果判断出在指定时间内收到新的节点信息，但节点状态没发生变化，则在 nodeStatusMap 中保存该节点的状态信息，并用 Node Controller 所在节点的系统时间作为探测时间，用上次节点信息中的节点状态变化时间作为该节点的状态变化时间。如果判断出在某一段时间 (gracePeriod) 内没有收到节点状态信息，则设置节点状态为“未知”(Unknown)，并且通过 API Server 保存节点状态。

(3) 逐个读取节点信息，如果节点状态变为非“就绪”状态，则将节点加入待删除队列，否则将节点从该队列中删除。如果节点状态为非“就绪”状态，且系统指定了 Cloud Provider，则 Node Controller 调用 Cloud Provider 查看节点，若发现节点故障，则删除 etcd 中的节点信息，并删除和该节点相关的 Pod 等资源的信息。

3.2.3 ResourceQuota Controller

作为完备的企业级的容器集群管理平台，Kubernetes 也提供了资源配置管理(ResourceQuota Controller)这一高级功能，资源配置管理确保了指定的资源对象在任何时候都不会超量占用系统物理资源，避免了由于某些业务进程的设计或实现的缺陷导致整个系统运行紊乱甚至意外宕机，对整个集群的平稳运行和稳定性有非常重要的作用。

目前 Kubernetes 支持如下三个层次的资源配置管理。

- (1) 容器级别，可以对 CPU 和 Memory 进行限制。
- (2) Pod 级别，可以对一个 Pod 内所有容器的可用资源进行限制。
- (3) Namespace 级别，为 Namespace (多租户) 级别的资源限制，包括：
 - ◎ Pod 数量；
 - ◎ Replication Controller 数量；
 - ◎ Service 数量；

- ◎ ResourceQuota 数量；
- ◎ Secret 数量；
- ◎ 可持有的 PV（Persistent Volume）数量。

Kubernetes 的配额管理是通过 Admission Control（准入控制）来控制的，Admission Control 当前提供了两种方式的配额约束，分别是 LimitRanger 与 ResourceQuota。其中 LimitRanger 作用于 Pod 和 Container 上，而 ResourceQuota 则作用于 Namespace 上，限定一个 Namespace 里的各类资源的使用总额。

如图 3.5 所示，如果在 Pod 定义中同时声明了 LimitRanger，则用户通过 API Server 请求创建或修改资源时，Admission Control 会计算当前配额的使用情况，如果不符合配额约束，则创建对象失败。对于定义了 ResourceQuota 的 Namespace，ResourceQuota Controller 组件则负责定期统计和生成该 Namespace 下的各类对象的资源使用总量，统计结果包括 Pod、Service、RC、Secret 和 Persistent Volume 等对象实例个数，以及该 Namespace 下所有 Container 实例所使用的资源量（目前包括 CPU 和内存），然后将这些统计结果写入 etcd 的 resourceQuotaStatusStorage 目录（resourceQuotas/status）中。写入 resourceQuotaStatusStorage 的内容包含 Resource 名称、配额值（ResourceQuota 对象中 spec.hard 域下包含的资源的值）、当前使用值（ResourceQuota Controller 统计出来的值）。随后这些统计信息被 Admission Control 使用，以确保相关 Namespace 下的资源配置总量不会超过 ResourceQuota 中的限定值。

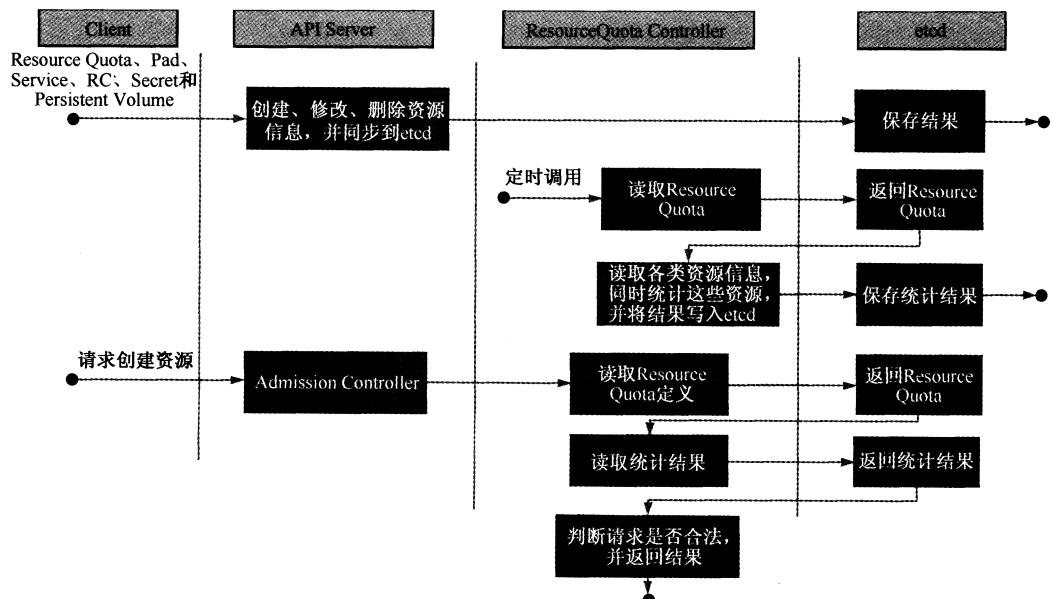


图 3.5 ResourceQuota Controller 流程图

3.2.4 Namespace Controller

用户通过 API Server 可以创建新的 Namespace 并保存在 etcd 中，Namespace Controller 定时通过 API Server 读取这些 Namespace 信息。如果 Namespace 被 API 标识为优雅删除（通过设置删除期限，即 `DeletionTimestamp` 属性被设置），则将该 Namespace 的状态设置成“Terminating”并保存到 etcd 中。同时 Namespace Controller 删除该 Namespace 下的 ServiceAccount、RC、Pod、Secret、PersistentVolume、ListRange、ResourceQuota 和 Event 等资源对象。

当 Namespace 的状态被设置成“Terminating”后，由 Admission Controller 的 `NamespaceLifecycle` 插件来阻止为该 Namespace 创建新的资源。同时，在 Namespace Controller 删除完该 Namespace 中的所有资源对象后，Namespace Controller 对该 Namespace 执行 `finalize` 操作，删除 Namespace 的 `spec.finalizers` 域中的信息。

如果 Namespace Controller 观察到 Namespace 设置了删除期限，同时 Namespace 的 `spec.finalizers` 域值是空的，那么 Namespace Controller 将通过 API Server 删除该 Namespace 资源。

3.2.5 Service Controller 与 Endpoint Controller

我们先说说 Endpoints Controller，在这之前，让我们先看看 Service、Endpoints 与 Pod 的关系，如图 3.6 所示，Endpoints 表示了一个 Service 对应的所有 Pod 副本的访问地址，而 Endpoints Controller 就是负责生成和维护所有 Endpoints 对象的控制器。

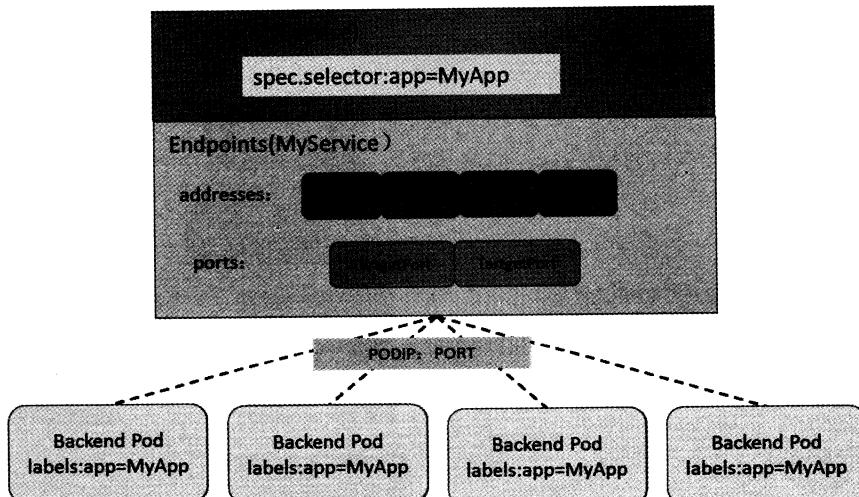


图 3.6 Service、Endpoint、Pod 的关系

它负责监听 Service 和对应的 Pod 副本的变化，如果监测到 Service 被删除，则删除和该 Service 同名的 Endpoints 对象；如果监测到新的 Service 被创建或者修改，则根据该 Service 信息获得相关的 Pod 列表，然后创建或者更新 Service 对应的 Endpoints 对象。如果监测到 Pod 的事件，则更新它所对应的 Service 的 Endpoints 对象（增加、删除或者修改对应的 Endpoint 条目）。

那么，Endpoints 对象是在哪里被使用的呢？答案是每个 Node 上的 kube-proxy 进程，kube-proxy 进程获取每个 Service 的 Endpoints，实现了 Service 的负载均衡功能。在后面的章节中我们会深入讲解这部分内容。

接下来我们说说 Service Controller 的作用，它其实是属于 Kubernetes 集群与外部的云平台之间的一个接口控制器。Service Controller 监听 Service 的变化，如果是一个 LoadBalancer 类型的 Service (`externalLoadBalancers=true`)，则 Service Controller 确保外部的云平台上该 Service 对应的 LoadBalancer 实例被相应地创建、删除及更新路由转发表（根据 Endpoints 的条目）。

3.3 Scheduler 原理分析

我们在前面深入分析了 Controller Manager 及它所包含的各个组件的运行机制。本节我们将继续对 Kubernetes 中负责 Pod 调度的重要功能模块——Kubernetes Scheduler 的工作原理和运行机制做深入分析。

Kubernetes Scheduler 在整个系统中承担了“承上启下”的重要功能，“承上”是指它负责接收 Controller Manager 创建的新 Pod，为其安排一个落脚的“家”——目标 Node；“启下”是指安置工作完成后，目标 Node 上的 kubelet 服务进程接管后继工作，负责 Pod 生命周期中的“下半生”。

具体来说，Kubernetes Scheduler 的作用是将待调度的 Pod (API 新创建的 Pod、Controller Manager 为补足副本而创建的 Pod 等) 按照特定的调度算法和调度策略绑定 (Binding) 到集群中的某个合适的 Node 上，并将绑定信息写入 etcd 中。在整个调度过程中涉及三个对象，分别是：待调度 Pod 列表、可用 Node 列表，以及调度算法和策略。简单地说，就是通过调度算法调度为待调度 Pod 列表的每个 Pod 从 Node 列表中选择一个最适合的 Node。

随后，目标节点上的 kubelet 通过 API Server 监听到 Kubernetes Scheduler 产生的 Pod 绑定事件，然后获取对应的 Pod 清单，下载 Image 镜像，并启动容器。完整的流程如图 3.7 所示。

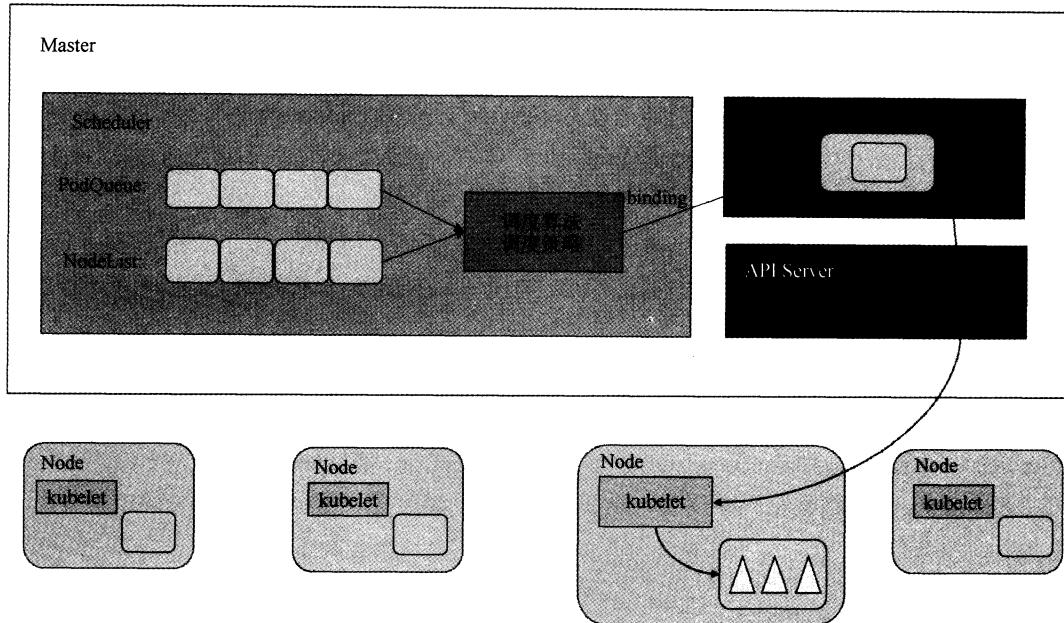


图 3.7 Scheduler 流程

Kubernetes Scheduler 当前提供的默认调度流程分为以下两步。

(1) 预选调度过程，即遍历所有目标 Node，筛选出符合要求的候选节点。为此，Kubernetes 内置了多种预选策略（xxx Predicates）供用户选择。

(2) 确定最优节点，在第 1 步的基础上，采用优选策略（xxx Priority）计算出每个候选节点的积分，积分最高者胜出。

Kubernetes Scheduler 的调度流程是通过插件方式加载的“调度算法提供者”(AlgorithmProvider)具体实现的。一个 AlgorithmProvider 其实就是包括了一组预选策略与一组优先选择策略的结构体，注册 AlgorithmProvider 的函数如下：

```
func RegisterAlgorithmProvider(name string, predicateKeys, priorityKeys
util.StringSet)
```

它包含三个参数：“name string”参数为算法名；“predicateKeys”参数为算法用到的预选策略集合；“priorityKeys”为算法用到的优选策略集合。

Scheduler 中可用的预选策略包含：NoDiskConflict、PodFitsResources、PodSelectorMatches、PodFitsHost、CheckNodeLabelPresence、CheckServiceAffinity 和 PodFitsPorts 策略等。其默认的 AlgorithmProvider 加载的预选策略 Predicates 包括：PodFitsPorts (PodFitsPorts)、PodFitsResources (PodFitsResources)、NoDiskConflict (NoDiskConflict)、MatchNodeSelector (PodSelectorMatches)

和 HostName（PodFitsHost），即每个节点只有通过前面提及的 5 个默认预选策略后，才能初步被选中，进入下一个流程。

下面列出的是对所有预选策略的详细说明。

1) NoDiskConflict

判断备选 Pod 的 GCEPersistentDisk 或 AWSElasticBlockStore 和备选的节点中已存在的 Pod 是否存在冲突。检测过程如下。

(1) 首先，读取备选 Pod 的所有 Volume 的信息（即 pod.Spec.Volumes），对每个 Volume 执行以下步骤进行冲突检测。

(2) 如果该 Volume 是 GCEPersistentDisk，则将 Volume 和备选节点上的所有 Pod 的每个 Volume 进行比较，如果发现相同的 GCEPersistentDisk，则返回 false，表明存在磁盘冲突，检查结束，反馈给调度器该备选节点不适合作为备选 Pod；如果该 Volume 是 AWSElasticBlockStore，则将 Volume 和备选节点上的所有 Pod 的每个 Volume 进行比较，如果发现相同的 AWSElasticBlockStore，则返回 false，表明存在磁盘冲突，检查结束，反馈给调度器该备选节点不适合备选 Pod。

(3) 如果检查完备选 Pod 的所有 Volume 均未发现冲突，则返回 true，表明不存在磁盘冲突，反馈给调度器该备选节点适合备选 Pod。

2) PodFitsResources

判断备选节点的资源是否满足备选 Pod 的需求，检测过程如下。

(1) 计算备选 Pod 和节点中已存在 Pod 的所有容器的需求资源（内存和 CPU）的总和。

(2) 获得备选节点的状态信息，其中包含节点的资源信息。

(3) 如果备选 Pod 和节点中已存在 Pod 的所有容器的需求资源（内存和 CPU）的总和，超出了备选节点拥有的资源，则返回 false，表明备选节点不适合备选 Pod，否则返回 true，表明备选节点适合备选 Pod。

3) PodSelectorMatches

判断备选节点是否包含备选 Pod 的标签选择器指定的标签。

(1) 如果 Pod 没有指定 spec.nodeSelector 标签选择器，则返回 true。

(2) 否则，获得备选节点的标签信息，判断节点是否包含备选 Pod 的标签选择器（spec.nodeSelector）所指定的标签，如果包含，则返回 true，否则返回 false。

4) PodFitsHost

判断备选 Pod 的 spec.nodeName 域所指定的节点名称和备选节点的名称是否一致，如果一

致，则返回 true，否则返回 false。

5) CheckNodeLabelPresence

如果用户在配置文件中指定了该策略，则 Scheduler 会通过 RegisterCustomFitPredicate 方法注册该策略。该策略用于判断策略列出的标签在备选节点中存在时，是否选择该备选节点。

- (1) 读取备选节点的标签列表信息。
- (2) 如果策略配置的标签列表存在于备选节点的标签列表中，且策略配置的 presence 值为 false，则返回 false，否则返回 true；如果策略配置的标签列表不存在于备选节点的标签列表中，且策略配置的 presence 值为 true，则返回 false，否则返回 true。

6) CheckServiceAffinity

如果用户在配置文件中指定了该策略，则 Scheduler 会通过 RegisterCustomFitPredicate 方法注册该策略。该策略用于判断备选节点是否包含策略指定的标签，或包含和备选 Pod 在相同 Service 和 Namespace 下的 Pod 所在节点的标签列表。如果存在，则返回 true，否则返回 false。

7) PodFitsPorts

判断备选 Pod 所用的端口列表中的端口是否在备选节点中已被占用，如果被占用，则返回 false，否则返回 true。

Scheduler 中的优选策略包含：LeastRequestedPriority、CalculateNodeLabelPriority 和 BalancedResourceAllocation 等。每个节点通过优先选择策略时都会算出一个得分，计算各项得分，最终选出得分值最大的节点作为优选的结果（也是调度算法的结果）。

下面是对所有优选策略的详细说明。

1) LeastRequestedPriority

该优选策略用于从备选节点列表中选出资源消耗最小的节点。

- (1) 计算出所有备选节点上运行的 Pod 和备选 Pod 的 CPU 占用量 totalMilliCPU。
- (2) 计算出所有备选节点上运行的 Pod 和备选 Pod 的内存占用量 totalMemory。
- (3) 计算每个节点的得分，计算规则大致如下。

NodeCpuCapacity 为节点 CPU 计算能力；NodeMemoryCapacity 为节点内存大小。

```
score=int(((nodeCpuCapacity-totalMilliCPU)*10)/ nodeCpuCapacity+((nodeMemoryCapacity-totalMemory)*10)/ nodeCpuMemory)/2
```

2) CalculateNodeLabelPriority

如果用户在配置文件中指定了该策略，则 scheduler 会通过 RegisterCustomPriorityFunction 方法注册该策略。该策略用于判断策略列出的标签在备选节点中存在时，是否选择该备选节点。

如果备选节点的标签在优选策略的标签列表中且优选策略的 presence 值为 true，或者备选节点的标签不在优选策略的标签列表中且优选策略的 presence 值为 false，则备选节点 score=10，否则备选节点 score=0。

3) BalancedResourceAllocation

该优选策略用于从备选节点列表中选出各项资源使用率最均衡的节点。

- (1) 计算出所有备选节点上运行的 Pod 和备选 Pod 的 CPU 占用量 totalMilliCPU。
- (2) 计算出所有备选节点上运行的 Pod 和备选 Pod 的内存占用量 totalMemory。
- (3) 计算每个节点的得分，计算规则大致如下。

NodeCpuCapacity 为节点 CPU 计算能力；NodeMemoryCapacity 为节点内存大小。

```
score= int(10-math.Abs(totalMilliCPU/nodeCpuCapacity-totalMemory/
nodeMemoryCapacity)*10)
```

3.4 kubelet 运行机制分析

在 Kubernetes 集群中，在每个 Node 节点（又称 Minion）上都会启动一个 kubelet 服务进程。该进程用于处理 Master 节点下发到本节点的任务，管理 Pod 及 Pod 中的容器。每个 kubelet 进程会在 API Server 上注册节点自身信息，定期向 Master 节点汇报节点资源的使用情况，并通过 cAdvisor 监控容器和节点资源。

3.4.1 节点管理

节点通过设置 kubelet 的启动参数“--register-node”，来决定是否向 API Server 注册自己。如果该参数的值为 true，那么 kubelet 将试着通过 API Server 注册自己。在自注册时，kubelet 启动时还包含下列参数。

- ◎ --api-servers: 告诉 kubelet API Server 的位置。
- ◎ --kubeconfig: 告诉 kubelet 在哪儿可以找到用于访问 API Server 的证书。
- ◎ --cloud-provider: 告诉 kubelet 如何从云服务商（IaaS）那里读取到和自己相关的元数据。

当前每个 kubelet 被授予创建和修改任何节点的权限。但是在实践中，它仅仅创建和修改自己。将来，我们计划限制 kubelet 的权限，仅允许它修改和创建其所在节点的权限。如果在集群运行过程中遇到集群资源不足的情况，则用户很容易通过添加机器及运用 kubelet 的自注册模式

来实现扩容。

在某些情况下，Kubernetes 集群中的某些 kubelet 没有选择自注册模式，用户需要自己去配置 Node 的资源信息，同时告知 Node 上的 kubelet API Server 的位置。集群管理者能够创建和修改节点信息。如果管理者希望手动创建节点信息，则通过设置 kubelet 的启动参数“`--register-node=false`”即可。

kubelet 在启动时通过 API Server 注册节点信息，并定时向 API Server 发送节点的新消息，API Server 在接收到这些信息后，将这些信息写入 etcd。通过 kubelet 的启动参数“`--node-status-update-frequency`”设置 kubelet 每隔多少时间向 API Server 报告节点状态，默认为 10 秒。

3.4.2 Pod 管理

kubelet 通过以下几种方式获取自身 Node 上所要运行的 Pod 清单。

(1) 文件：kubelet 启动参数“`--config`”指定的配置文件目录下的文件（默认目录为“`/etc/kubernetes/manifests/`”）。通过`--file-check-frequency` 设置检查该文件目录的时间间隔，默认为 20 秒。

(2) HTTP 端点（URL）：通过“`--manifest-url`”参数设置。通过`--http-check-frequency` 设置检查该 HTTP 端点数据的时间间隔，默认为 20 秒。

(3) API Server：kubelet 通过 API Server 监听 etcd 目录，同步 Pod 列表。

所有以非 API Server 方式创建的 Pod 都叫作 Static Pod。kubelet 将 Static Pod 的状态汇报给 API Server，API Server 为该 Static Pod 创建一个 Mirror Pod 和其相匹配。Mirror Pod 的状态将真实反映 Static Pod 的状态。当 Static Pod 被删除时，与之相对应的 Mirror Pod 也会被删除。在本章中我们只讨论通过 API Server 获得 Pod 清单的方式。kubelet 通过 API Server Client 使用 Watch 加 List 的方式监听“`/registry/nodes/$当前节点的名称`”和“`/registry/pods`”目录，将获取的信息同步到本地缓存中。

kubelet 监听 etcd，所有针对 Pod 的操作将会被 kubelet 监听到。如果发现有新的绑定到本节点的 Pod，则按照 Pod 清单的要求创建该 Pod。

如果发现本地的 Pod 被修改，则 kubelet 会做出相应的修改，比如删除 Pod 中的某个容器时，则通过 Docker Client 删除该容器。

如果发现删除本节点的 Pod，则删除相应的 Pod，并通过 Docker Client 删除 Pod 中的容器。kubelet 读取监听到的信息，如果是创建和修改 Pod 任务，则做如下处理。

(1) 为该 Pod 创建一个数据目录。

- (2) 从 API Server 读取该 Pod 清单。
- (3) 为该 Pod 挂载外部卷 (External Volume)。
- (4) 下载 Pod 用到的 Secret。
- (5) 检查已经运行在节点中的 Pod，如果该 Pod 没有容器或 Pause 容器 (“kubernetes/pause” 镜像创建的容器) 没有启动，则先停止 Pod 里所有容器的进程。如果在 Pod 中有需要删除的容器，则删除这些容器。
- (6) 用 “kubernetes/pause” 镜像为每个 Pod 创建一个容器。该 Pause 容器用于接管 Pod 中所有其他容器的网络。每创建一个新的 Pod，kubelet 都会先创建一个 Pause 容器，然后创建其他容器。“kubernetes/pause” 镜像大概为 200KB，是一个非常小的容器镜像。
- (7) 为 Pod 中的每个容器做如下处理。
 - ◎ 为容器计算一个 hash 值，然后用容器的名字去查询对应 Docker 容器的 hash 值。若查找到容器，且两者的 hash 值不同，则停止 Docker 中容器的进程，并停止与之关联的 Pause 容器的进程；若两者相同，则不做任何处理。
 - ◎ 如果容器被终止了，且容器没有指定的 restartPolicy (重启策略)，则不做任何处理。
 - ◎ 调用 Docker Client 下载容器镜像，调用 Docker Client 运行容器。

3.4.3 容器健康检查

Pod 通过两类探针来检查容器的健康状态。一个是 LivenessProbe 探针，用于判断容器是否健康，告诉 kubelet 一个容器什么时候处于不健康的状态。如果 LivenessProbe 探针探测到容器不健康，则 kubelet 将删除该容器，并根据容器的重启策略做相应的处理。如果一个容器不包含 LivenessProbe 探针，那么 kubelet 认为该容器的 LivenessProbe 探针返回的值永远是 “Success”；另一类是 ReadinessProbe 探针，用于判断容器是否启动完成，且准备接收请求。如果 ReadinessProbe 探针检测到失败，则 Pod 的状态将被修改。Endpoint Controller 将从 Service 的 Endpoint 中删除包含该容器所在 Pod 的 IP 地址的 Endpoint 条目。

kubelet 定期调用容器中的 LivenessProbe 探针来诊断容器的健康状况。LivenessProbe 包含以下三种实现方式。

- (1) ExecAction: 在容器内部执行一个命令，如果该命令的退出状态码为 0，则表明容器健康。
- (2) TCPSocketAction: 通过容器的 IP 地址和端口号执行 TCP 检查，如果端口能被访问，则表明容器健康。

(3) **HTTPGetAction**: 通过容器的 IP 地址和端口号及路径调用 HTTP Get 方法，如果响应的状态码大于等于 200 且小于等于 400，则认为容器状态健康。

LivenessProbe 探针包含在 Pod 定义的 spec.containers.{某个容器}中。下面的例子展示了两种 Pod 中容器健康检查的方式：HTTP 检查和容器命令执行检查。下面所列的内容实现了通过容器命令执行检查：

```
livenessProbe:  
  exec:  
    command:  
      - cat  
      - /tmp/health  
  initialDelaySeconds: 15  
  timeoutSeconds: 1
```

kubelet 在容器中执行“cat /tmp/health”命令，如果该命令返回的值为 0，则表明容器处于健康状态，否则表明容器处于不健康状态。

下面所列的内容实现了容器的 HTTP 检查：

```
livenessProbe:  
  httpGet:  
    path: /healthz  
    port: 8080  
  initialDelaySeconds: 15  
  timeoutSeconds: 1
```

kubelet 发送一个 HTTP 请求到本地主机和端口及指定的路径，来检查容器的健康状况。

3.4.4 cAdvisor 资源监控

在 Kubernetes 集群中如何监控资源的使用情况？

在 Kubernetes 集群中，应用程序的执行情况可以在不同的级别上监测到，这些级别包括：容器、Pod、Service 和整个集群。作为 Kubernetes 集群的一部分，Kubernetes 希望提供给用户详细的各个级别的资源使用信息，这将使用户能够深入地了解应用的执行情况，并找到应用中可能的瓶颈。Heapster 项目为 Kubernetes 提供了一个基本的监控平台，它是集群级别的监控和事件数据集成器(Aggregator)。Heapster 作为 Pod 运行在 Kubernetes 集群中，和运行在 Kubernetes 集群中的其他应用相似。Heapster Pod 通过 kubelet（运行在节点上的 Kubernetes 代理）发现所有运行在集群中的节点，并查看来自这些节点的资源使用状况信息。kubelet 通过 cAdvisor 获取其所在节点及容器的数据，Heapster 通过带着关联标签的 Pod 分组这些信息，这些数据被推到一个可配置的后端，用于存储和可视化展示。当前支持的后端包括 InfluxDB（with Grafana for

Visualization) 和 Google Cloud Monitoring。

cAdvisor 是一个开源的分析容器资源使用率和性能特性的代理工具。它是因为容器而产生的，因此自然支持 Docker 容器。在 Kubernetes 项目中，cAdvisor 被集成到 Kubernetes 代码中。cAdvisor 自动查找所有在其所在节点上的容器，自动采集 CPU、内存、文件系统和网络使用的统计信息。cAdvisor 通过它所在节点机的 Root 容器，采集并分析该节点机的全面使用情况。

在大部分 Kubernetes 集群中，cAdvisor 通过它所在节点机的 4194 端口暴露一个简单的 UI。图 3.8 是 cAdvisor 的一个截图。

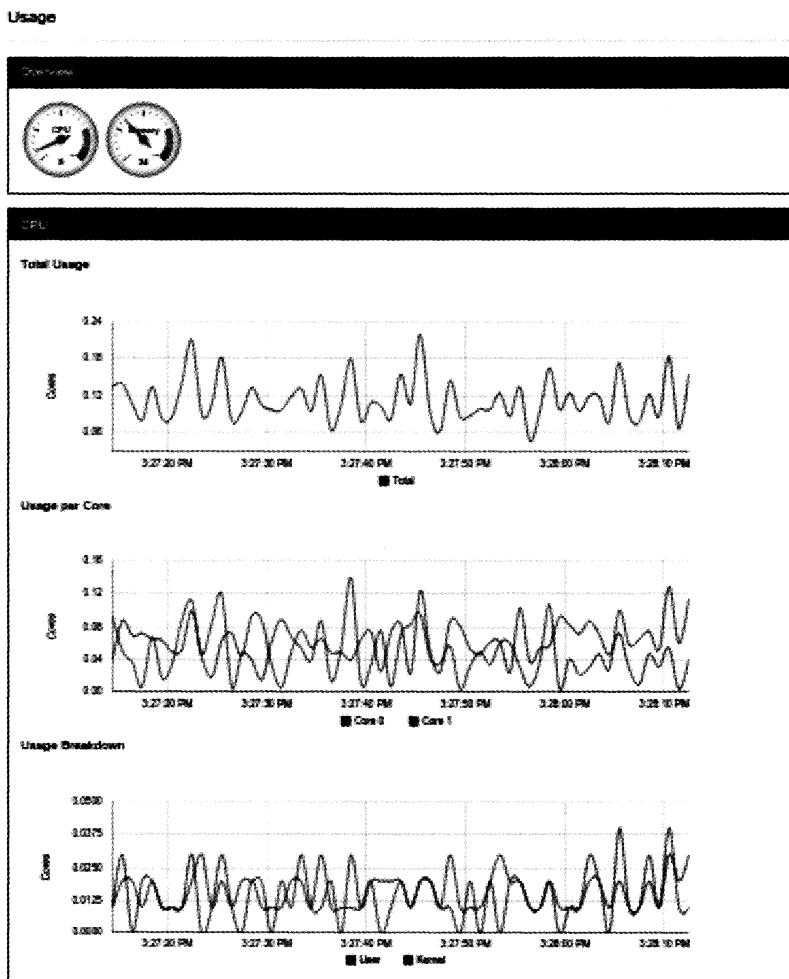


图 3.8 cAdvisor 的一个 UI

kubelet 作为连接 Kubernetes Master 和各节点机之间的桥梁，管理运行在节点机上的 Pod 和容器。kubelet 将每个 Pod 转换成它的成员容器，同时从 cAdvisor 获取单独的容器使用统计信息，然后通过该 REST API 暴露这些聚合后的 Pod 资源使用的统计信息。

3.5 kube-proxy 运行机制分析

我们在前面已经了解到，为了支持集群的水平扩展、高可用性，Kubernetes 抽象出了 Service 的概念。Service 是对一组 Pod 的抽象，它会根据访问策略（如负载均衡策略）来访问这组 Pod。

Kubernetes 在创建服务时会为服务分配一个虚拟的 IP 地址，客户端通过访问这个虚拟的 IP 地址来访问服务，而服务则负责将请求转发到后端的 Pod 上。这不就是一个反向代理吗？不错，这就是一个反向代理。但是，它和普通的反向代理有一些不同：首先它的 IP 地址是虚拟的，想从外面访问还需要一些技巧；其次是它的部署和启停是 Kubernetes 统一自动管理的。

Service 在很多情况下只是一个概念，而真正将 Service 的作用落实的是背后的 kube-proxy 服务进程。只有理解了 kube-proxy 的原理和机制，我们才能真正理解 Service 背后的实现逻辑。

在 Kubernetes 集群的每个 Node 上都会运行一个 kube-proxy 服务进程，这个进程可以看作 Service 的透明代理兼负载均衡器，其核心功能是将到某个 Service 的访问请求转发到后端的多个 Pod 实例上。对每一个 TCP 类型的 Kubernetes Service，kube-proxy 都会在本地 Node 上建立一个 SocketServer 来负责接收请求，然后均匀发送到后端某个 Pod 的端口上，这个过程默认采用 Round Robin 负载均衡算法。另外，Kubernetes 也提供通过修改 Service 的 service.spec.sessionAffinity 参数的值来实现会话保持特性的定向转发，如果设置的值为“ClientIP”，则将来自同一个 ClientIP 的请求都转发到同一个后端 Pod 上。

此外，Service 的 Cluster IP 与 NodePort 等概念是 kube-proxy 服务通过 Iptables 的 NAT 转换实现的，kube-proxy 在运行过程中动态创建与 Service 相关的 Iptables 规则，这些规则实现了 Cluster IP 及 NodePort 的请求流量重定向到 kube-proxy 进程上对应服务的代理端口的功能。由于 Iptables 机制针对的是本地的 kube-proxy 端口，所以每个 Node 上都要运行 kube-proxy 组件，这样一来，在 Kubernetes 集群内部，我们可以在任意 Node 上发起对 Service 的访问请求。

综上所述，由于 kube-proxy 的作用，在 Service 的调用过程中客户端无须关心后端有几个 Pod，中间过程的通信、负载均衡及故障恢复都是透明的，如图 3.9 所示。

访问 Service 的请求，不论是用 Cluster IP + TargetPort 的方式，还是用节点机 IP+ NodePort 的方式，都被节点机的 Iptables 规则重定向到 kube-proxy 监听 Service 服务代理端口。kube-proxy 接收到 Service 的访问请求后，会如何选择后端的 Pod 呢？

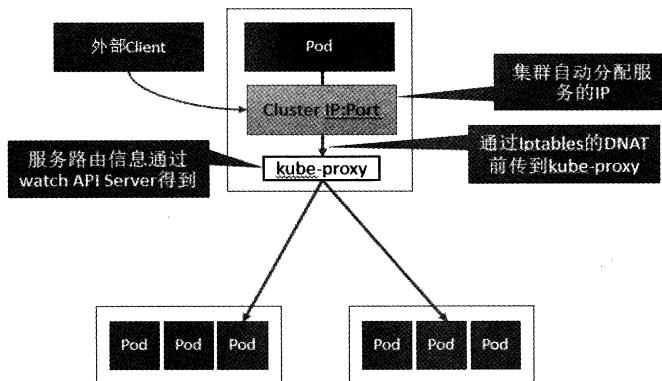


图 3.9 Service 的负载均衡转发规则

首先，目前 kube-proxy 的负载均衡器只支持 Round Robin 算法。Round Robin 算法按照成员列表逐个选取成员，如果一轮循环完，便从头开始下一轮，如此循环往复。kube-proxy 的负载均衡器在 Round Robin 算法的基础上还支持 Session 保持。如果 Service 在定义中指定了 Session 保持，则 kube-proxy 接收请求时会从本地内存中查找是否存在来自该请求 IP 的 affinityState 对象，如果存在该对象，且 Session 没有超时，则 kube-proxy 将请求转向该 affinityState 所指向的后端 Pod。如果本地存在没有来自该请求 IP 的 affinityState 对象，则按照 Round Robin 算法为该请求挑选一个 Endpoint，并创建一个 affinityState 对象，记录请求的 IP 和指向的 Endpoint。后面的请求就会“黏连”到这个创建好的 affinityState 对象上，这就实现了客户端 IP 会话保持的功能。

接下来我们深入分析 kube-proxy 的实现细节。

kube-proxy 通过查询和监听 API Server 中 Service 与 Endpoints 的变化，为每个 Service 都建立了一个“服务代理对象”，并自动同步。服务代理对象是 kube-proxy 程序内部的一种数据结构，它包括一个用于监听此服务请求的 SocketServer，SocketServer 的端口是随机选择的一个本地空闲端口。此外，kube-proxy 内部也创建了一个负载均衡器——LoadBalancer，LoadBalancer 上保存了 Service 到对应的后端 Endpoint 列表的动态转发路由表，而具体的路由选择则取决于 Round Robin 负载均衡算法及 Service 的 Session 会话保持（SessionAffinity）这两个特性。

针对发生变化的 Service 列表，kube-proxy 会逐个处理。下面是具体的处理流程。

(1) 如果该 Service 没有设置集群 IP (ClusterIP)，则不做任何处理，否则，获取该 Service 的所有端口定义列表 (spec.ports 域)。

(2) 逐个读取服务端口定义列表中的端口信息，根据端口名称、Service 名称和 Namespace 判断本地是否已经存在对应的服务代理对象，如果不存在则新建；如果存在并且 Service 端口被修改过，则先删除 Iptables 中和该 Service 端口相关的规则，关闭服务代理对象，然后走新建流

程，即为该 Service 端口分配服务代理对象并为该 Service 创建相关的 Iptables 规则。

(3) 更新负载均衡器组件中对应 Service 的转发地址列表，对于新建的 Service，确定转发时的会话保持策略。

(4) 对于已经删除的 Service 则进行清理。

而针对 Endpoint 的变化，kube-proxy 会自动更新负载均衡器中对应 Service 的转发地址列表。

下面讲解 kube-proxy 针对 Iptables 所做的一些细节操作。

kube-proxy 在启动时和监听到 Service 或 Endpoint 的变化后，会在本机 Iptables 的 NAT 表中添加 4 条规则链。

(1) KUBE-PORALS-CONTAINER：从容器中通过 Service Cluster IP 和端口号访问 Service 的请求。

(2) KUBE-PORALS-HOST：从主机中通过 Service Cluster IP 和端口号访问 Service 的请求。

(3) KUBE-NODEPORT-CONTAINER：从容器中通过 Service 的 NodePort 端口号访问 Service 的请求。

(4) KUBE-NODEPORT-HOST：从主机中通过 Service 的 NodePort 端口号访问 Service 的请求。

此外，kube-proxy 在 Iptables 中为每个 Service 创建由 Cluster IP + Service 端口到 kube-proxy 所在主机 IP + Service 代理服务所监听的端口的转发规则。转发规则的包匹配规则部分 (CRETIRIA) 如下所示：

```
-m comment --comment $SERVICESTRING -p $PROTOCOL -m $PROTOCOL --dport $DESTPORT  
-d $DESTIP
```

其中，“-m comment --comment”表示匹配规则使用 Iptables 的显式扩展的注释功能；“\$SERVICESTRING”为注释的内容；“-p \$PROTOCOL -m \$PROTOCOL --dport \$DESTPORT -d \$DESTIP”表示协议为“\$PROTOCOL”且目标地址和端口为“\$DESTIP”和“\$DESTPORT”的包，其中，“\$PROTOCOL”可以为 TCP 或 UDP，“\$DESTIP”和“\$DESTPORT”为 Service 的 Cluster IP 和 TargetPort。

对于转发规则的跳转部分 (-j 部分)，如果请求来自本地容器，且 Service 代理服务监听的是所有的接口（例如 IPv4 的地址为 0.0.0.0），则跳转部分如下所示：

```
-j REDIRECT --to-ports $proxyPort
```

其表示该规则的功能是实现数据包的端口重定向，重定向到\$proxyPort 端口（Service 代理服务监听的端口）；否则，跳转部分如下所示：

```
-j DNAT --to-destination proxyIP:proxyPort
```

表示该规则的功能是实现数据包转发，数据包的目的地址变为“proxyIP:proxyPort”（即

Service 代理服务所在的 IP 地址和端口，这些地址和端口都会被替换成实际的地址和端口)。

如果 Service 类型为 NodePort，则 kube-proxy 在 Iptables 中除了添加上面提及的规则，还会为每个 Service 创建由 NodePort 端口到 kube-proxy 所在主机 IP + Service 代理服务所监听的端口的转发规则。转发规则的包匹配规则部分 (CRETIRIA) 如下所示：

```
-m comment --comment $SERVICESTRING -p $PROTOCOL -m $PROTOCOL --dport $NODEPORT
```

上面所列的内容用于匹配目的端口为 “\$NODEPORT” 的包。

转发规则的跳转部分 (-j 部分) 和前面提及的跳转规则一致。

最后，我们以本书第 2 章的 Hello World 为例，看看 kube-proxy 为 redis-master 服务所生成的 Iptables 转发规则：

```
$ iptables-save | grep redis-master
-A KUBE-PORTALS-CONTAINER -d 10.254.208.57/32 -p tcp -m comment --comment
"default/redis-master:" -m tcp --dport 6379 -j REDIRECT --to-ports 42872
-A KUBE-PORTALS-HOST -d 10.254.208.57/32 -p tcp -m comment --comment
"default/redis-master:" -m tcp --dport 6379 -j DNAT --to-destination
192.168.1.130:42872
```

可以看到，对 “redis-master” Service 的 6379 端口的访问将会被转发到物理机的 42872 端口上。而 42872 端口就是 kube-proxy 为这个 Service 打开的随机本地端口。

最后，给出本节的一个总结性的示意图，如图 3.10 所示。

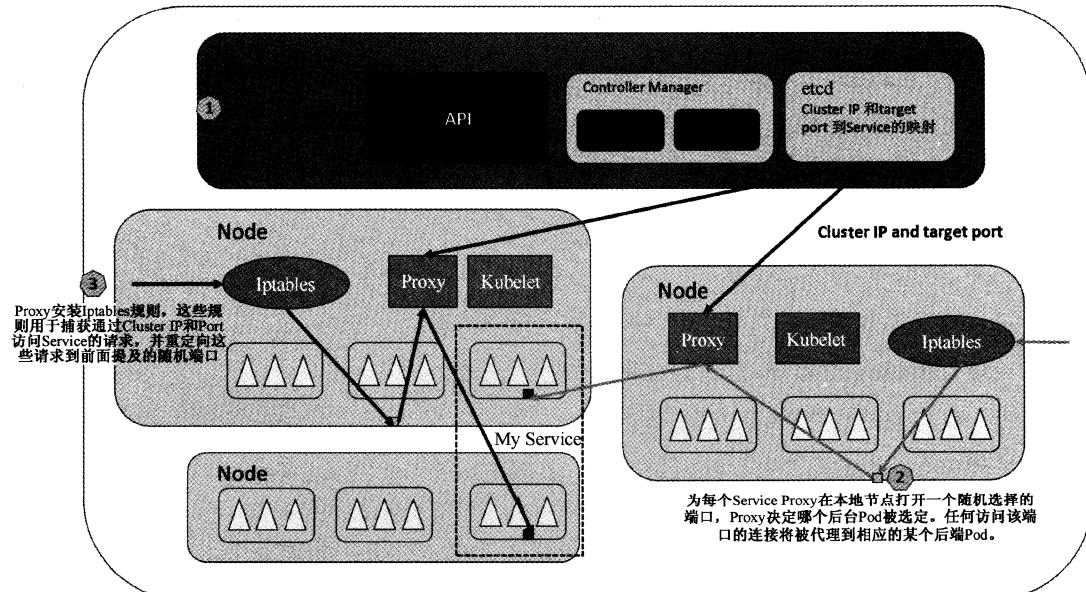


图 3.10 kube-proxy 工作原理示意图

3.6 深入分析集群安全机制

Kubernetes 通过一系列机制来实现集群的安全控制，其中包括 API Server 的认证授权、准入控制机制及保护敏感信息的 Secret 机制等。集群的安全性必须考虑如下几个目标。

- (1) 保证容器与其所在的宿主机的隔离。
- (2) 限制容器给基础设施及其他容器带来消极影响的能力。
- (3) 最小权限原则——合理限制所有组件的权限，确保组件只执行它被授权的行为，通过限制单个组件的能力来限制它所能到达的权限范围。
- (4) 明确组件间边界的划分。
- (5) 划分普通用户和管理员的角色。
- (6) 在必要的时候允许将管理员权限赋给普通用户。
- (7) 允许拥有“Secret”数据（Keys、Certs、Passwords）的应用在集群中运行。

下面分别从 Authentication、Authorization、Admission Control、Secret 和 Service Account 等方面来说明集群的安全机制。

3.6.1 API Server 认证

我们知道，Kubernetes 集群中所有资源的访问和变更都是通过 Kubernetes API Server 的 REST API 来实现的，所以集群安全的关键点就在于如何识别并认证客户端身份（Authentication），以及随后访问权限的授权（Authorization）这两个关键问题，本节我们讲解前一个问题。

我们知道，Kubernetes 集群提供了 3 种级别的客户端身份认证方式。

- ◎ 最严格的 HTTPS 证书认证：基于 CA 根证书签名的双向数字证书认证方式。
- ◎ HTTP Token 认证：通过一个 Token 来识别合法用户。
- ◎ HTTP Base 认证：通过用户名+密码的方式认证。

首先，我们说说 HTTPS 证书认证的原理。

这里需要有一个 CA 证书，我们知道 CA 是 PKI 系统中通信双方都信任的实体，被称为可信第三方（Trusted Third Party, TTP）。CA 作为可信第三方的重要条件之一就是 CA 的行为具有非否认性。作为第三方而不是简单的上级，就必须能让信任者有追究自己责任的能力。CA 通过证书证实他人的公钥信息，证书上有 CA 的签名。用户如果因为信任证书而有了损失，则证书可以作为有效的证据用于追究 CA 的法律责任。正是因为 CA 承担责任的承诺，所以 CA

也被称为可信第三方。在很多情况下，CA与用户是相互独立的实体，CA作为服务提供方，有可能因为服务质量问题（例如，发布的公钥数据有错误）而给用户带来损失。在证书中绑定了公钥数据和相应私钥拥有者的身份信息，并带有CA的数字签名；证书中也包含了CA的名称，以便于依赖方找到CA的公钥，验证证书上的数字签名。

CA认证涉及诸多概念，比如根证书、自签名证书、密钥、私钥、加密算法及HTTPS等，本书大致讲述SSL协议的流程，有助于对CA认证和Kubernetes CA认证的配置过程的理解。

如图3.11所示，SSL双向认证大概包含下面几个步骤。

(1) HTTPS通信双方的服务器端向CA机构申请证书，CA机构是可信的第三方机构，它可以是一个公认的权威的企业，也可以是企业自身。企业内部系统一般都用企业自身的认证系统。CA机构下发根证书、服务端证书及私钥给申请者。

(2) HTTPS通信双方的客户端向CA机构申请证书，CA机构下发根证书、客户端证书及私钥给申请者。

(3) 客户端向服务器端发起请求，服务端下发服务端证书给客户端。客户端接收到证书后，通过私钥解密证书，并利用服务器端证书中的公钥认证证书信息比较证书里的消息，例如域名和公钥与服务器刚刚发送的相关消息是否一致，如果一致，则客户端认可这个服务器的合法身份。

(4) 客户端发送客户端证书给服务器端，服务端接收到证书后，通过私钥解密证书，获得客户端证书公钥，并用该公钥认证证书信息，确认客户端是否合法。

(5) 客户端通过随机密钥加密信息，并发送加密后的信息给服务端。服务器端和客户端协商好加密方案后，客户端会产生一个随机的密钥，客户端通过协商好的加密方案，加密该随机密钥，并发送该随机密钥到服务器端。服务器端接收这个密钥后，双方通信的所有内容都通过该随机密钥加密。

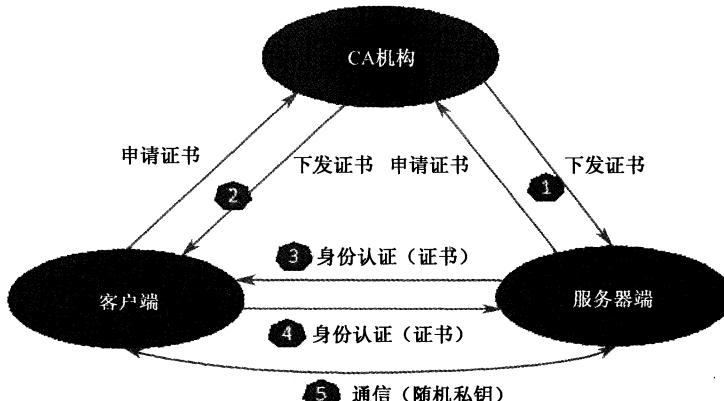


图3.11 CA认证流程

如上所述是双向认证 SSL 协议的具体通信过程，这种情况要求服务器和用户双方都有证书。单向认证 SSL 协议不需要客户拥有 CA 证书，对于上面的步骤，只需将服务器端验证客户证书的过程去掉，以及在协商对称密码方案和对称通话密钥时，服务器发送给客户的是没有加过密的（这并不影响 SSL 过程的安全性）密码方案。

其次，我们来看看 HTTP Token 的认证原理。

HTTP Token 的认证是用一个很长的特殊编码方式的并且难以被模仿的字符串——Token 来表明客户身份的一种方式。在通常情况下，Token 是一个很复杂的字符串，比如我们用私钥签名一个字符串后的数据就可以当作一个 Token。此外，每个 Token 对应一个用户名，存储在 API Server 能访问的一个文件中。当客户端发起 API 调用请求时，需要在 HTTP Header 里放入 Token，这样一来，API Server 就能识别合法用户和非法用户了。

最后，我们说说 HTTP Base 认证。

我们知道，HTTP 协议是无状态的，浏览器和 Web 服务器之间可以通过 Cookie 来进行身份识别。桌面应用程序（比如新浪桌面客户端、SkyDrive 客户端、命令行程序）一般不会使用 Cookie，那么它们与 Web 服务器之间是如何进行身份识别的呢？这就用到了 HTTP Base 认证，这种认证方式是把“用户名+冒号+密码”用 BASE64 算法进行编码后的字符串放在 HTTP Request 中的 Header Authorization 域里发送给服务端，服务端收到后进行解码，获取用户名及密码，然后进行用户身份的鉴权过程。

3.6.2 API Server 授权

对合法用户进行授权（Authorization）并且随后在用户访问时进行鉴权，是权限与安全系统的重要一环。简单地说，授权就是授予不同的用户不同的访问权限，API Server 目前支持以下几种授权策略（通过 API Server 的启动参数“`--authorization_mode`”设置）。

- ◎ AlwaysDeny。
- ◎ AlwaysAllow。
- ◎ ABAC。

其中，AlwaysDeny 表示拒绝所有的请求，该配置一般用于测试；AlwaysAllow 表示接收所有的请求，如果集群不需要授权流程，则可以采用该策略，这也是 Kubernetes 的默认配置；ABAC（Attribute-Based Access Control）为基于属性的访问控制，表示使用用户配置的授权规则去匹配用户的请求。

为了简化授权的复杂度，对于 ABAC 模式的授权策略，Kubernetes 仅有下面四个基本属性。

- ◎ 用户名（代表一个已经被认证的用户的字符型用户名）。
- ◎ 是否是只读请求（REST的GET操作是只读的）。
- ◎ 被访问的是哪一类资源，例如访问Pod资源/api/v1/namespaces/default/pods。
- ◎ 被访问对象所属的Namespace。

当我们为API Server启用ABAC模式时，需要指定授权策略文件的路径和名字（--authorization_policy_file=SOME_FILENAME），授权策略文件里的每一行都是一个Map类型的JSON对象，被称为“访问策略对象”，我们可以通过设置“访问策略对象”中的如下属性来确定具体的授权行为。

- ◎ user（用户名）：为字符串类型，该字符串类型的用户名来源于Token文件或基本认证文件中的用户名字段的值。
- ◎ readonly（只读标识）：为布尔类型，当它的值为true时，表明该策略允许GET请求通过。
- ◎ resource（资源）：为字符串类型，来自于URL的资源，例如“Pods”。
- ◎ namespace（命名空间）：为字符串类型，表明该策略允许访问某个Namespace的资源。

例如，我们要实现如下访问控制。

- (1) 允许用户alice做任何事情
- (2) kubelet只能访问Pod的只读API。
- (3) kubelet能读和写Event对象。
- (4) 用户bob只能访问myNamespace中的Pod的只读API。

则满足上述要求的授权策略文件的内容写法如下：

```
{"user": "alice"}
{"user": "kubelet", "resource": "pods", "readonly": true}
{"user": "kubelet", "resource": "events"}
{"user": "bob", "resource": "pods", "readonly": true, "ns": "myNamespace"}
```

当客户端发起API Server调用时，API Server内部要先进行用户认证，接下来执行用户鉴权流程，鉴权流程通过之前提到的“授权策略”来决定一个API调用是否合法。当API Server接收到请求后，会读取该请求中的数据，生成一个“访问策略对象”，如果该请求中不带某些属性（如Namespace），则这些属性的值将根据属性类型的不同，设置不同的默认值（例如为字符串类型的属性设置一个空字符串；为布尔类型的属性设置false；为数值类型的属性设置0）。然后用这个“访问策略对象”和授权策略文件中的所有“访问策略对象”逐条匹配，如果至少有一个策略对象被匹配上，则该请求将被鉴权通过，否则终止API调用流程，并返回客户端错误调用码。

3.6.3 Admission Control 准入控制

突破了之前所说的认证和鉴权两道关口之后，客户端的调用请求就能够得到 API Server 的真正响应了吗？答案是：不能！这个请求还需要通过 Admission Control 所控制的一个“准入控制链”的层层考验，官方标准的“关卡”有近十个之多，而且能自定义扩展！笔者忽然在想，如果在幼儿园的时候，老师就告诉我们长大后还要读小学，参加中考、高考、公司面试、职称考试，等等，我们还会天天去幼儿园吗？

Admission Control 配备有一个“准入控制器”的列表，发送给 API Server 的任何请求都需要通过列表中每个准入控制器的检查，检查不通过，则 API Server 拒绝此调用请求。此外，准入控制器还能够修改请求参数以完成一些自动化的任务，比如 ServiceAccount 这个控制器。当前可配置的准入控制器如下。

- ◎ **AlwaysAdmit**: 允许所有请求。
- ◎ **AlwaysPullImages**: 在启动容器之前总是去下载镜像，相当于在每个容器的配置项 `imagePullPolicy=Always`。
- ◎ **AlwaysDeny**: 禁止所有请求，一般用于测试。
- ◎ **DenyExecOnPrivileged**: 它会拦截所有想在 Privileged Container 上执行命令的请求。如果你的集群支持 Privileged Container，你又希望限制用户在这些 Privileged Container 上执行命令，那么强烈推荐你使用它。
- ◎ **ServiceAccount**: 这个 plug-in 将 serviceAccounts 实现了自动化，默认启用，如果你想要使用 ServiceAccount 对象，那么强烈推荐你使用它，后面讲述 ServiceAccount 的章节会详细说明其作用。
- ◎ **SecurityContextDeny**: 这个插件将使用了 SecurityContext 的 Pod 中定义的选项全部失效。SecurityContext 在 Container 中定义了操作系统级别的安全设定（uid、gid、capabilities、SELinux 等）。
- ◎ **ResourceQuota**: 用于配额管理目的，作用于 Namespace 上，它会观察所有的请求，确保在 namespace 上的配额不会超标。推荐在 Admission Control 参数列表中这个插件排最后一个。
- ◎ **LimitRanger**: 用于配额管理，作用于 Pod 与 Container 上，确保 Pod 与 Container 上的配额不会超标。
- ◎ **NamespaceExists** (已过时): 对所有请求校验 namespace 是否已存在，如果不存在则拒绝请求。已合并至 NamespaceLifecycle。

- ◎ NamespaceAutoProvision（已过时）：对所有请求校验 namespace，如果不存在则自动创建该 namespace，推荐使用 NamespaceLifecycle。
- ◎ NamespaceLifecycle：如果尝试在一个不存在的 namespace 中创建资源对象，则该创建请求将被拒绝。当删除一个 namespace 时，系统将会删除该 namespace 中的所有对象，包括 Pod、Service 等。

在 API Server 上设置--admission-control 参数，即可定制我们需要的准入控制链，如果启用多种准入控制选项，则建议的设置（含加载顺序）如下：

```
--admission-control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota
```

大部分准入控制器都比较容易理解，我们接下来着重介绍 SecurityContextDeny、ResourceQuota 及 LimitRanger 这三个准入控制器。

1) SecurityContextDeny

Security Context 是运用于容器的操作系统安全设置（uid、gid、capabilities、SELinux role 等）。Admission Control 的 SecurityContextDeny 插件的作用是，禁止创建设置了 Security Context 的 Pod，例如包含下面这些配置项的 Pod：

```
spec.containers.securityContext.seLinuxOptions
spec.containers.securityContext.runAsUser
```

2) ResourceQuota

准入控制器 ResourceQuota 不仅能够限制某个 Namespace 中创建资源的数量，而且能够限制某个 Namespace 中被 Pod 所请求的资源总量。该准入控制器和资源对象 ResourceQuota 一起实现了资源配置管理。

3) LimitRanger

准入控制器 LimitRanger 的作用类似于上面的 ResourceQuota 控制器，针对 Namespace 资源的每个个体（Pod 与 Container 等）的资源配置。该插件和资源对象 LimitRange 一起实现资源限制管理。

3.6.4 Service Account

Service Account 也是一种账号，但它并不是给 Kubernetes 的集群的用户（系统管理员、运维人员、租户用户等）使用的，而是给运行在 Pod 里的进程用的，它为 Pod 里的进程提供必要的身份证明。

在继续学习之前，请回忆一下本章前面所说的 API Server 的认证一节。

我们知道，正常情况下，为了确保 Kubernetes 集群的安全，API Server 都会对客户端进行身份认证，认证失败的客户端无法进行 API 调用。此外，Pod 中访问 Kubernetes API Server 服务的时候，是以 Service 方式访问服务名为 kubernetes 的这个服务的，而 kubernetes 服务又只在 HTTPS 安全端口 443 上提供服务，那么如何进行身份认证呢？这的确是个谜，因为 Kubernetes 的官方文档并没有清楚说明这个问题。

通过查看官方源码，我们发现这是在用一种类似 HTTP Token 的新的认证方式——Service Account Auth，Pod 中的客户端调用 kubernetes API 的时候，在 HTTP Header 中传递了一个 Token 字符串，这类似于之前提到的 HTTP Token 认证方式，但又有以下几个不同点。

- ◎ 这个 Token 的内容来自 Pod 里指定路径下的一个文件（/run/secrets/kubernetes.io/serviceaccount/token），这种 Token 是动态生成的，确切地说，是由 Kubernetes Controller 进程用 API Server 的私钥（--service-account-private-key-file 指定的私钥）签名生成的一个 JWT Secret。
- ◎ 官方提供的客户端 REST 框架代码里，通过 HTTPS 方式与 API Server 建立连接后，会用 Pod 里指定路径下的一个 CA 证书（/run/secrets/kubernetes.io/serviceaccount/ca.crt）验证 API Server 发来的证书，验证是否是被 CA 证书签名的合法证书。
- ◎ API Server 收到这个 Token 以后，采用自己的私钥（实际是使用参数 service-account-key-file 指定的私钥，如果此参数没有设置，则默认采用 tls-private-key-file 指定的参数，即自己的私钥）对 Token 进行合法性验证。

明白了认证原理，我们接下来继续分析上面认证过程中所涉及的 Pod 中的以下三个文件。

- ◎ /run/secrets/kubernetes.io/serviceaccount/token。
- ◎ /run/secrets/kubernetes.io/serviceaccount/ca.crt。
- ◎ /run/secrets/kubernetes.io/serviceaccount/namespace（客户端采用这里指定的 namespace 作为参数调用 Kubernetes API）。

这三个文件由于参与到 Pod 进程与 API Server 认证的过程中，起到了类似 Secret（私密凭据）的作用，所以它们被称为 Kubernetes Secret 对象。Secret 从属于 Service Account 资源对象，属于 Service Account 的一部分，一个 Service Account 对象里面可以包括多个不同的 Secret 对象，分别用于不同目的的认证活动。

下面我们通过运行一些命令来加深我们对 Service Account 与 Secret 的直观认识。

首先，查看系统中的 Service Account 对象，我们看到有一个名为 default 的 Service Account 对象，包含一个名为 default-token-77oyg 的 Secret，这个 Secret 同时是“Mountable secrets”，表明它是需要被 Mount 到 Pod 上的：

```
# kubectl describe serviceaccounts
Name:      default
Namespace: default
Labels:    <none>
Image pull secrets: <none>
Mountable secrets:  default-token-77oyg
Tokens:    default-token-77oyg
```

接下来，我们看看 default-token-77oyg 都有什么内容：

```
# kubectl describe secrets default-token-77oyg
Name:      default-token-77oyg
Namespace: default
Labels:    <none>
Annotations: kubernetes.io/service-account.name=default
            kubernetes.io/service-account.uid=3e5b99c0-432c-11e6-b45c-000c29dc2102

Type:  kubernetes.io/service-account-token

Data
=====
token:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJrdWJlcmt1dGVzL3NlcnZpY2VhY2Nv
dW50Iiwia3ViZXJuZXRLcy5pb3VudC9uYW1lc3BhY2UiOijkZWhdWx0Iiwia3Vi
ZXJuZXRLcy5pb3VudC9uYW1lc3BhY2UiOijkZWhdWx0Iiwia3Vi
ZXJuZXRLcy5pb3VudC9uYW1lc3BhY2UiOijkZWhdWx0Iiwia3Vi
LCJrdWJlcmt1dGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW11IjoiZGVmYXVs
dCIasImt1YmVybmV0ZXMuaw8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjN1NWI5
OWMwLTQzMmMtMTF1Ni1iNDVjLTAWMGMyOWRjmjEwMiIsInN1YiI6InN5c3RlbTpzZXJ2aWN1YWNjb3Vu
dDpkZWhdWx0OmR1ZmF1bHQifQ.MFsBrYmTLMB55X3UGfO_pADP6FSSqgHb0SxGJtTsJnY-ze2vFc8Qd
07bVdmQffBnkHgLWh1KIpR_EvvJTRP538uovgcA_QGN9yIMEdqIfQC2wfnLFuk10a80dSH4uzayBb50
yI7gJWXWbXn6u0wAGMneiTktCvzGfR4q-p19Jjh5qNPiUdJ0NhjsJJSAclhdNK40Xt0gMHdNNyPEmpgk
60w2cM7DRb6ifiSo05cTeLyv1TpIBMvcQy4sYedCEL2cJ20BwcSo4-1Dev9rdxr50dtgCvo60xbPF7R
cWwjggUMLYO3YCi07WmQNdmxWHJkwvBtkWhzdvuFCpHeWANA
ca.crt:      1115 bytes
namespace: 7 bytes
```

从上面的输出信息中我们看到，default-token-77oyg 包括三个数据项，分别是 token、ca.crt、namespace。联想到“Mountable secrets”的标记，以及之前看到的 Pod 中的三个文件的文件名，你可能恍然大悟，原来是这么一回事：每个 Namespace 下有一个名为 default 的默认的 Service Account 对象，这个 Service Account 里面有一个名为 Tokens 的可以当作 Volume 一样被 Mount 到 Pod 里的 Secret，当 Pod 启动时，这个 Secret 会自动被 Mount 到 Pod 的指定目录下，用来协助完成 Pod 中的进程访问 API Server 时的身份鉴权过程。

如图 3.12 所示，一个 Service Account 可以包括多个 Secret 对象。

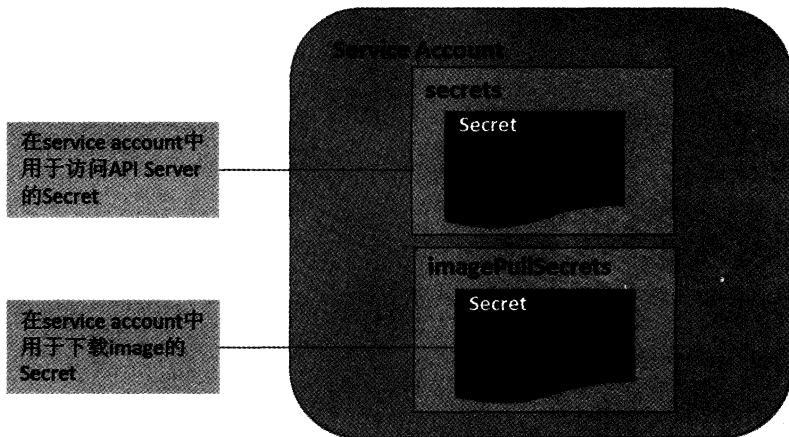


图 3.12 Service Account 中的 Secret

- (1) 名为 Tokens 的 Secret 用于访问 API Server 的 Secret，也被称为 Service Account Secret。
- (2) 名为 Image pull secrets 的 Secret 用于下载容器镜像时的认证过程，通常镜像库运行在 Insecure 模式下，所以这个 Secret 为空。
- (3) 用户自定义的其他 Secret，用于用户的进程。

如果一个 Pod 在定义时没有指定 spec.serviceAccountName 属性，则系统会自动为其赋值为“default”，即大家都使用同一个 Namespace 下默认的 Service Account。如果某个 Pod 需要使用非 default 的 Service Account，则需要在定义时指定：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mycontainter
      image: nginx:v1
  serviceAccountName: myserviceaccount
```

Kubernetes 之所以要创建两套独立的账号系统，原因如下。

- ◎ User 账号是给人用的，Service Account 是给 Pod 里的进程使用的，面向的对象不同。
- ◎ User 账号是全局性的，Service Account 则属于某个具体的 Namespace。
- ◎ 通常来说，User 账号是与后端的用户数据库同步的，创建一个新用户通常要走一套复杂的业务流程才能实现，Service Account 的创建则需要极轻量级的实现方式，集群管理员可以很容易为某些特定任务创建一个 Service Account。

- ◎ 对于这两种不同的账号，其审计要求通常不同。
- ◎ 对于一个复杂的系统来说，多个组件通常拥有各种账号的配置信息，Service Account 是 Namespace 隔离的，可以针对组件进行一对一的定义，同时具备很好的“便携性”。
接下来，我们深入分析 Service Account 与 Secret 相关的一些运行机制。

前面的 Controller Manager 原理分析一节中，我们知道 Controller manager 创建了 ServiceAccount Controller 与 Token Controller 两个安全相关的控制器。其中 ServiceAccount Controller 一直监听 Service Account 和 Namespace 的事件，如果一个 Namespace 中没有 default Service Account，那么 ServiceAccount Controller 就会为该 Namespace 创建一个默认（default）的 Service Account，这就是我们之前看到每个 Namespace 下都有一个名为 default 的 Service Account 的原因了。

如果 Controller manager 进程在启动时指定了 API Server 私钥（service-account-private-key-file 参数），那么 Controller manager 会创建 Token Controller。Token Controller 也监听 Service Account 的事件，如果发现新创建的 Service Account 里没有对应的 Service Account Secret，则会用 API Server 私钥创建一个 Token（JWT Token），并用该 Token、CA 证书及 Namespace 名称等三个信息产生一个新的 Secret 对象，然后放入刚才的 Service Account 中；如果监听到的事件是删除 Service Account 事件，则自动删除与该 Service Account 相关的所有 Secret。此外，Token Controller 对象同时监听 Secret 的创建、修改和删除事件，并根据事件的不同做不同的处理。

当我们在 API Server 的鉴权过程中启用了 Service Account 类型的准入控制器，即在 kube-apiserver 启动参数中包括下面的内容时：

```
--admission_control=ServiceAccount
```

则针对 Pod 新增或修改的请求，Service Account 准入控制器会验证 Pod 里的 Service Account 是否合法。

- (1) 如果 spec.serviceAccount 域没有被设置，则 Kubernetes 默认认为其指定名字为 default 的 Service account。
- (2) 如果 Pod 的 spec.serviceAccount 域指定了 default 以外的 Service Account，而该 Service Account 没有事先被创建，则该 Pod 操作失败。
- (3) 如果在 Pod 中没有指定 “ImagePullSecrets”，那么这个 spec.serviceAccount 域指定的 Service Account 的 “ImagePullSecrets” 会被加入该 Pod 中。
- (4) 给 Pod 添加一个特殊的 Volume，在该 Volume 中包含 Service Account Secret 中的 Token，并将 Volume 挂载到 Pod 中所有容器的指定目录下（/var/run/secrets/kubernetes.io/serviceaccount）。

综上所述，Service Account 的正常工作离不开以下几个控制器。

- (1) Admission Controller。

- (2) Token Controller。
- (3) ServiceAccount Controller。

3.6.5 Secret 私密凭据

上一节我们提到 Secret 对象，Secret 的主要作用是保管私密数据，比如密码、OAuth Tokens、SSH Keys 等信息。将这些私密信息放在 Secret 对象中比直接放在 Pod 或 Docker Image 中更安全，也更便于使用和分发。

下面的例子用于创建一个 Secret：

```
secrets.yaml:  
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
type: Opaque  
data:  
  password: dmFsdWUtMg0K  
  username: dmFsdWUtMQ0K  
  
# kubectl create -f secrets.yaml
```

在上面的例子中，data 域的各子域的值必须为 BASE64 编码值，其中 password 域和 username 域 BASE64 编码前的值分别为“value-1”和“value-2”。

一旦 Secret 被创建，则可以通过下面的三种方式使用它。

- (1) 在创建 Pod 时，通过为 Pod 指定 Service Account 来自动使用该 Secret。
- (2) 通过挂载该 Secret 到 Pod 来使用它。
- (3) Docker 镜像下载时使用，通过指定 Pod 的 spec.ImagePullSecrets 来引用它。

第 1 种使用方式主要用在 API Server 鉴权方面，之前我们提到过。下面的例子展示了第 2 种使用方式：将一个 Secret 通过挂载的方式添加到 Pod 的 Volume 中。

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: mypod  
  namespace: myns  
spec:  
  containers:  
    - name: mycontainer  
      image: redis
```

```

volumeMounts:
- name: foo
  mountPath: "/etc/foo"
  readOnly: true
volumes:
- name: foo
  secret:
    secretName: mysecret
  
```

其结果如图 3.13 所示。

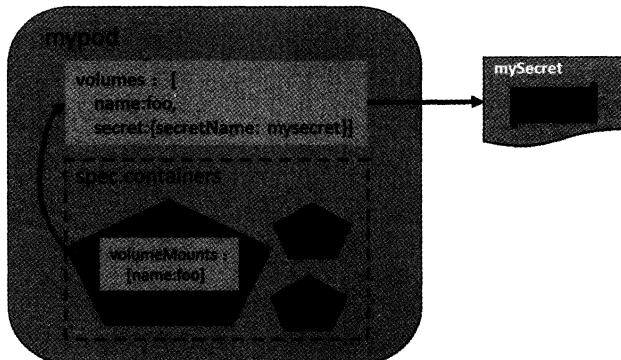


图 3.13 挂载 Secret 到 Pod

第3种使用方式的使用流程如下。

(1) 执行 login 命令，登录私有 Registry：

```
# docker login localhost:5000
```

输入用户名和密码，如果是第1次登录系统，则会创建新用户，相关信息会写入`~/.dockercfg`文件中。

(2) 用 BASE64 编码 dockercfg 的内容：

```
# cat ~/.dockercfg | base64
```

(3) 将上一步命令的输出结果作为 Secret 的“`data.dockercfg`”域的内容，由此来创建一个 Secret：

```

image-pull-secret.yaml:
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
data:
  .dockercfg: eyAiaHR0cHM6Ly9pbmRleC5kb2NrZXIuaW8vdjEvIjogeyAiYXV0aCI6ICJab
UZyWlhCaGMzTjNiM0prTVRJSyIsICJ1bWFpbCI6ICJqZG9lQGV4YW1wbGUuY29tIiB9IH0K
  type: kubernetes.io/dockercfg
  
```

```
# kubectl create -f image-pull-secret.yaml
```

(4) 在创建 Pod 时引用该 Secret：

```
pods.yaml:  
apiVersion: v1  
kind: Pod  
metadata:  
  name: mypod2  
spec:  
  containers:  
    - name: foo  
      image: janedoe/awesomeapp:v1  
  imagePullSecrets:  
    - name: myregistrykey
```

```
$ kubectl create -f pods.yaml
```

其结果如图 3.14 所示。

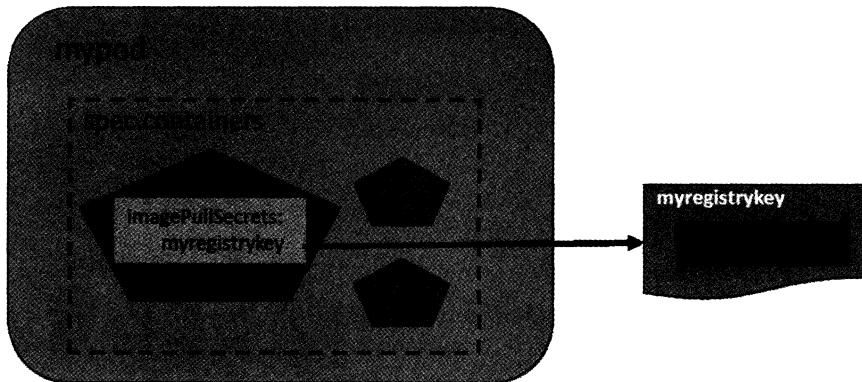


图 3.14 imagePullSecret 引用 Secret

每个单独的 Secret 大小不能超过 1MB，Kubernetes 不鼓励创建大尺寸的 Secret，因为如果使用大尺寸的 Secret，则将大量占用 API Server 和 kubelet 的内存。当然，创建许多小的 Secret 也能耗尽 API Server 和 kubelet 的内存。

在使用 Mount 方式挂载 Secret 时，Container 中 Secret 的“data”域的各个域的 Key 值作为目录中的文件，Value 值被 BASE64 编码后存储在相应的文件中。前面的例子中创建的 Secret，被挂载到一个叫作 mycontainer 的 Container 中，在该 Container 中可通过相应的查询命令查看所生成的文件和文件中的内容，如下所示：

```
$ ls /etc/foo/  
username  
password
```

```
$ cat /etc/foo/username
value-1
$ cat /etc/foo/password
value-2
```

通过上面的例子可以得出如下结论：我们可以通过 Secret 保管其他系统的敏感信息（比如数据库的用户名和密码），并以 Mount 的方式将 Secret 挂载到 Container 中，然后通过访问目录中的文件的方式获取该敏感信息。当 Pod 被 API Server 创建时，API Server 不会校验该 Pod 引用的 Secret 是否存在。一旦这个 Pod 被调度，则 kubelet 将试着获取 Secret 的值。如果 Secret 不存在或暂时无法连接到 API Server，则 kubelet 将按一定的时间间隔定期重试获取该 Secret，并发送一个 Event 来解释 Pod 没有启动的原因。一旦 Secret 被 Pod 获取，则 kubelet 将创建并 Mount 包含 Secret 的 Volume。只有所有 Volume 被 Mount 后，Pod 中的 Container 才会被启动。在 kubelet 启动 Pod 中的 Container 后，Container 中的和 Secret 相关的 Volume 将不会被改变，即使 Secret 本身被修改了。为了使用更新后的 Secret，必须删除旧的 Pod，并重新创建一个新的 Pod，因此更新 Secret 的流程和部署一个新的 Image 是一样的。

3.7 网络原理

关于 Kubernetes 网络，我们通常有这些问题需要回答，如图 3.15 所示。

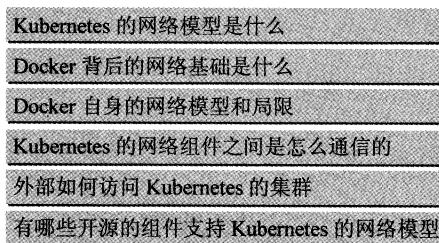


图 3.15 Kubernetes 的常见问题

在本节我们分别回答这些问题，然后通过一个具体的实验来将这些相关的知识串联成一个整体。

3.7.1 Kubernetes 网络模型

Kubernetes 网络模型设计的一个基本原则是：每个 Pod 都拥有一个独立的 IP 地址，而且假定所有 Pod 都在一个可以直接连通的、扁平的网络空间中。所以不管它们是否运行在同一个 Node（宿主机）中，都要求它们可以直接通过对方的 IP 进行访问。设计这个原则的原因是，用户不

需要额外考虑如何建立 Pod 之间的连接，也不需要考虑将容器端口映射到主机端口等问题。

实际上在 Kubernetes 的世界里，IP 是以 Pod 为单位进行分配的。一个 Pod 内部的所有容器共享一个网络堆栈（实际上就是一个网络命名空间，包括它们的 IP 地址、网络设备、配置等都是共享的）。按照这个网络原则抽象出来的一个 Pod 一个 IP 的设计模型也被称作 IP-per-Pod 模型。

由于 Kubernetes 的网络模型假设 Pod 之间访问时使用的是对方 Pod 的实际地址，所以一个 Pod 内部的应用程序看到的自己的 IP 地址和端口与集群内其他 Pod 看到的一样。它们都是 Pod 实际分配的 IP 地址（从 docker0 上分配的）。将 IP 地址和端口在 Pod 内部和外部都保持一致，我们可以不使用 NAT 来进行转换，地址空间也自然是平的。Kubernetes 的网络之所以这么设计，主要原因就是可以兼容过去的应用。当然，我们使用 Linux 命令“ip addr show”也能看到这些地址，和程序看到的没有什么区别。所以这种 IP-per-Pod 的方案很好地利用了现有的各种域名解析和发现机制。

一个 Pod 一个 IP 的模型还有另外一层含义，那就是同一个 Pod 内的不同容器将会共享一个网络命名空间，也就是说同一个 Linux 网络协议栈。这就意味着同一个 Pod 内的容器可以通过 localhost 来连接对方的端口。这种关系和同一个 VM 内的进程之间的关系是一样的，看起来 Pod 内的容器之间的隔离性降低了，而且 Pod 内不同容器之间的端口是共享的，没有所谓的私有端口的概念了。如果你的应用必须要使用一些特定的端口范围，那么你也可以为这些应用单独创建一些 Pod。反之，对那些没有特殊需要的应用，这样做的好处是 Pod 内的容器是共享部分资源的，通过共享资源互相通信显然更加容易和高效。针对这些应用，虽然损失了可接受范围内的部分隔离性，但也是值得的。

IP-per-Pod 模式和 Docker 原生的通过动态端口映射方式实现的多节点访问模式有什么区别呢？主要区别是后者的动态端口映射会引入端口管理的复杂性，而且访问者看到的 IP 地址和端口与服务提供者实际绑定的不同（因为 NAT 的缘故，它们都被映射成新的地址或端口了），这也会引起应用配置的复杂化。同时，标准的 DNS 等名字解析服务也不适用了。甚至服务注册和发现机制都将受到挑战，因为在端口映射情况下，服务自身很难知道自己对外暴露的真实的服务 IP 和端口。而外部应用也无法通过服务所在容器的私有 IP 地址和端口来访问服务。

总的来说，IP-per-Pod 模型是一个简单的兼容性较好的模型。从该模型的网络的端口分配、域名解析、服务发现、负载均衡、应用配置和迁移等角度来看，Pod 都能够被看作一台独立的“虚拟机”或“物理机”。

按照这个网络抽象原则，Kubernetes 对网络有什么前提和要求呢？

Kubernetes 对集群的网络有如下要求：

- (1) 所有容器都可以在不用 NAT 的方式下同别的容器通信；

- (2) 所有节点都可以在不用 NAT 的方式下同所有容器通信，反之亦然；
- (3) 容器的地址和别人看到的地址是同一个地址。

这些基本的要求意味着并不是只要两台机器运行 Docker，Kubernetes 就可以工作了。具体的集群网络实现必须保障上述基本要求，原生的 Docker 网络目前还不能很好地支持这些要求。

实际上，这些对网络模型的要求并没有降低整个网络系统的复杂度。如果你的程序原来在 VM 上运行，而那些 VM 拥有独立 IP，并且它们之间可以直接透明地通信，那么 Kubernetes 的网络模型就和 VM 使用的网络模型是一样的。所以使用这种模型可以很容易地将已有的应用程序从 VM 或者物理机迁移到容器上。

当然，谷歌设计 Kubernetes 的一个主要运行基础就是其云环境 GCE（Google Compute Engine），在 GCE 下这些网络要求都是默认支持的。另外，常见的其他公有云服务商如亚马逊等，在它们的公有云计算环境下也是默认支持这个模型的。

由于部署私有云的场景会更普遍，所以在私有云中运行 Kubernetes+Docker 集群之前，就需要自己搭建出符合 Kubernetes 要求的网络环境。现在的开源世界有很多开源组件可以帮助我们打通 Docker 容器和容器之间的网络，实现 Kubernetes 要求的网络模型。当然每种方案都有适合的场景，我们要根据自己的实际需要进行选择。在后面的章节中会对常见的开源方案进行介绍。

Kubernetes 的网络依赖于 Docker，Docker 的网络又离不开 Linux 操作系统内核特性的支持，所以我们有必要先深入了解 Docker 背后的网络原理和基础知识。接下来我们一起深入学习一些必要的 Linux 网络知识。

3.7.2 Docker 的网络基础

Docker 本身的技术依赖于近年 Linux 内核虚拟化技术的发展，所以 Docker 对 Linux 内核的特性有很强的依赖。这里将 Docker 使用到的与 Linux 网络有关的主要技术进行简要介绍，这些技术包括如下几种，如图 3.16 所示。



图 3.16 Docker 使用到的与 Linux 网络有关的主要技术

1. 网络的命名空间

为了支持网络协议栈的多个实例，Linux 在网络栈中引入了网络命名空间（Network Namespace），这些独立的协议栈被隔离到不同的命名空间中。处于不同命名空间的网络栈是完全隔离的，彼此之间无法通信，就好像两个“平行宇宙”。通过这种对网络资源的隔离，就能在一个宿主机上虚拟多个不同的网络环境。而 Docker 也正是利用了网络的命名空间特性，实现了不同容器之间网络的隔离。

在 Linux 的网络命名空间内可以有自己独立的路由表及独立的 Iptables/Netfilter 设置来提供包转发、NAT 及 IP 包过滤等功能。

为了隔离出独立的协议栈，需要纳入命名空间的元素有进程、套接字、网络设备等。进程创建的套接字必须属于某个命名空间，套接字的操作也必须在命名空间内进行。同样，网络设备也必须属于某个命名空间。因为网络设备属于公共资源，所以可以通过修改属性实现在命名空间之间移动。当然，是否允许移动和设备的特征有关。

让我们稍微深入 Linux 操作系统内部，看它是如何实现网络命名空间的，这也会对理解后面的概念有帮助。

1) 网络命名空间的实现

Linux 的网络协议栈是十分复杂的，为了支持独立的协议栈，相关的这些全局变量都必须修改为协议栈私有。最好的办法就是让这些全局变量成为一个 Net Namespace 变量的成员，然后为协议栈的函数调用加入一个 Namespace 参数。这就是 Linux 实现网络命名空间的核心。

同时，为了保证对已经开发的应用程序及内核代码的兼容性，内核代码隐式地使用了命名空间内的变量。我们的程序如果没有对命名空间的特殊需求，那么不需要写额外的代码，网络命名空间对应用程序而言是透明的。

在建立了新的网络命名空间，并将某个进程关联到这个网络命名空间后，就出现了类似于如图 3.17 所示的内核数据结构，所有网站栈变量都放入了网络命名空间的数据结构中。这个网络命名空间是属于它的进程组私有的，和其他进程组不冲突。

新生成的私有命名空间只有回环 lo 设备（而且是停止状态），其他设备默认都不存在，如果我们需要，则要一一手工建立。Docker 容器中的各类网络栈设备都是 Docker Daemon 在启动时自动创建和配置的。

所有的网络设备（物理的或虚拟接口、桥等在内核里都叫作 Net Device）都只能属于一个命名空间。当然，通常物理的设备（连接实际硬件的设备）只能关联到 root 这个命名空间中。虚拟的网络设备（虚拟的以太网接口或者虚拟网口对）则可以被创建并关联到一个给定的命名空间中，而且可以在这些命名空间之间移动。

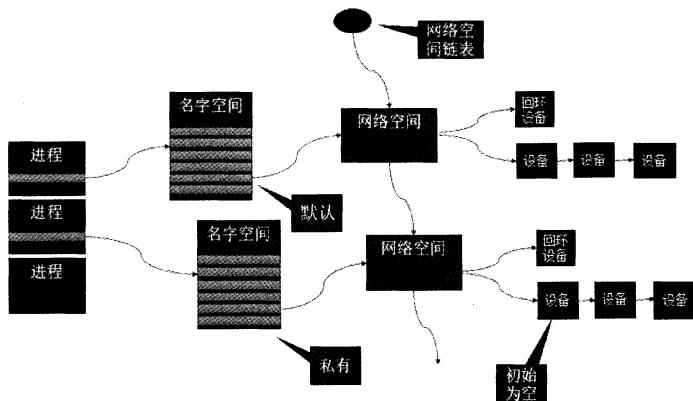


图 3.17 命名空间内核结构

前面我们提到，由于网络命名空间代表的是一个独立的协议栈，所以它们之间是相互隔离的，彼此无法通信，在协议栈内部都看不到对方。那么有没有办法打破这种限制，让处于不同命名空间的网络相互通信，甚至和外部的网络进行通信呢？答案就是“Veth 设备对”。“Veth 设备对”的一个重要作用就是打通互相看不到的协议栈之间的壁垒，它就像一个管子，一端连着这个网络命名空间的协议栈，一端连着另一个网络命名空间的协议栈。所以如果想在两个命名空间之间进行通信，就必须有一个 Veth 设备对。后面我们会介绍如何操作 Veth 设备对来打通不同命名空间之间的网络。

2) 网络命名空间操作

下面列举一些网络命名空间的操作。

我们可以使用 Linux iproute2 系列配置工具中的 IP 命令来操作网络命名空间。注意，这个命令需要由 root 用户运行。

创建一个命名空间：

```
ip netns add <name>
```

在命名空间内执行命令：

```
ip netns exec <name> <command>
```

如果想执行多个命令，则可以先进入内部的 sh，然后执行：

```
ip netns exec <name> bash
```

之后就是在新的命名空间内进行操作了。退出到外面的命名空间时，请输入“exit”。

3) 网络命名空间的一些技巧

操作网络命名空间时的一些实用技巧如下。

我们可以在不同的网络命名空间之间转移设备，例如下面会提到的 Veth 设备对的转移。因为一个设备只能属于一个命名空间，所以转移后在这个命名空间内就看不到这个设备了。具体哪些设备能够转移到不同的命名空间呢？在设备里面有一个重要的属性：NETIF_F_NETNS_LOCAL，如果这个属性为“on”，则不能转移到其他命名空间内。Veth 设备属于可以转移的设备，而很多其他设备如 lo 设备、vxlan 设备、ppp 设备、bridge 设备等都是不可以转移的。至于将无法转移的设备移动到别的命名空间的操作，则会得到无效参数的错误提示。

```
# ip link set br0 netns ns1
RTNETLINK answers: Invalid argument
```

如何知道这些设备是否可以转移呢？可以使用 ethtool 工具查看：

```
# ethtool -k br0
netns-local: on [fixed]
```

netns-local 的值是 on，就说明不可以转移，否则可以。

2. Veth 设备对

引入 Veth 设备对是为了在不同的网络命名空间之间进行通信，利用它可以直接将两个网络命名空间连接起来。由于要连接两个网络命名空间，所以 Veth 设备都是成对出现的，很像一对以太网卡，并且中间有一根直连的网线。既然是一对网卡，那么我们将其中一端称为另一端的 peer。在 Veth 设备的一端发送数据时，它会将数据直接发送到另一端，并触发另一端的接收操作。

整个 Veth 的实现非常简单，有兴趣的读者可以参考源代码“drivers/net/veth.c”的实现。图 3.18 是 Veth 设备对的示意图。

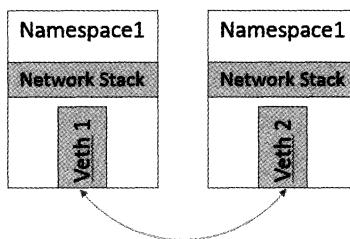


图 3.18 eth 设备对示意图

1) Veth 设备对的操作命令

接下来看看如何创建 Veth 设备对，如何连接到不同的命名空间，并设置它们的地址，让它们通信。

创建 Veth 设备对：

```
ip link add veth0 type veth peer name veth1
```

创建后，可以查看 veth 设备对的信息。使用 ip link show 命令查看所有网络接口：

```
# ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    Link/loopback: 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1677736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP mode DEFAULT qlen 1000
    link/ether 00:0c:29:cf:1a:2e brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state UP
mode DEFAULT
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
19: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether 7e:4a:ae:41:a3:65 brd ff:ff:ff:ff:ff:ff
20: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether ea:da:85:a3:75:8a brd ff:ff:ff:ff:ff:ff
```

看到了吧，有两个设备生成了，一个是 veth0，它的 peer 是 veth1。

现在这两个设备都在自己的命名空间内，那怎么能行呢？好了，如果将 Veth 看作有两个头的网线，那么我们将另一个头甩给另一个命名空间吧：

```
ip link set veth1 netns netns1
```

这时可在外面这个命名空间内看两个设备的情况：

```
# ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    Link/loopback: 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1677736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP mode DEFAULT qlen 1000
    link/ether 00:0c:29:cf:1a:2e brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state UP
mode DEFAULT
    link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
20: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether ea:da:85:a3:75:8a brd ff:ff:ff:ff:ff:ff
```

只剩一个 veth0 设备了，已经看不到另一个设备了，另一个设备已经转移到另一个网络命名空间了。

在 netns1 网络命名空间中可以看到 veth1 设备了，符合预期。

```
# ip netns exec netns1 ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    Link/loopback: 00:00:00:00:00:00 brd 00:00:00:00:00:00
19: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether 7e:4a:ae:41:a3:65 brd ff:ff:ff:ff:ff:ff
```

现在看到的结果是，两个不同的命名空间各自有一个 Veth 的“网线头”，各显示为一个 Device（在 Docker 的实现里面，它除了将 Veth 放入容器内，还将它的名字改成了 eth0，简直以假乱真，

你以为它是一个本地网卡吗)。

现在可以通信了吗？不行，因为它们还没有任何地址，现在我们来给它们分配 IP 地址吧：

```
ip netns exec netns1 ip addr add 10.1.1.1/24 dev veth1  
ip addr add 10.1.1.2/24 dev veth0
```

再启动它们：

```
ip netns exec netns1 ip link set dev veth1 up  
ip link set dev veth0 up
```

现在两个网络命名空间可以互相通信了：

```
# ping 10.1.1.1  
PING 10.1.1.1 (10.1.1.1) 56(84) bytes of data.  
64 bytes from 10.1.1.1: icmp_seq=1 ttl=64 time=0.035 ms  
64 bytes from 10.1.1.1: icmp_seq=2 ttl=64 time=0.096 ms  
^C  
--- 10.1.1.1 ping statistics ---  
2 packets transmitted, 2 received, 0% packet loss, time 1001ms  
rtt min/avg/max/mdev = 0.035/0.065/0.096/0.031 ms
```

```
# ip netns exec netns1 ping 10.1.1.2  
PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data.  
64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=0.045 ms  
64 bytes from 10.1.1.2: icmp_seq=2 ttl=64 time=0.105 ms  
^C  
--- 10.1.1.2 ping statistics ---  
2 packets transmitted, 2 received, 0% packet loss, time 1000ms  
rtt min/avg/max/mdev = 0.045/0.075/0.105/0.030 ms
```

至此，两个网络命名空间之间就完全通了。

至此我们就能够理解 Veth 设备对的原理和用法了。在 Docker 内部，Veth 设备对也是联系容器到外面的重要设备，离开它是不行的。

2) Veth 设备对如何查看对端

我们在操作 Veth 设备对的时候有一些实用技巧，如下所示。

一旦将 Veth 设备对的 peer 端放入另一个命名空间，我们在本命名空间内就看不到它了。那么我们怎么知道这个 Veth 对的对端在哪里呢，也就是说它到底连接到哪个别的命名空间呢？可以使用 ethtool 工具来查看（当网络命名空间特别多的时候，这可不是一件很容易的事情）。

首先我们在一个命名空间中查询 Veth 设备对端接口在设备列表中的序列号：

```
ip netns exec netns1 ethtool -S veth1  
NIC statistics:  
    peer_ifindex: 5
```

得知另一端的接口设备的序列号是 5，我们再到另一个命名空间中查看序列号 5 代表什么设备：

```
ip netns exec netns2 ip link | grep 5      <-- 我们只关注序列号是 5 的设备  
veth0
```

好了，我们现在就找到下标为 5 的设备了，它是 veth0，它的另一端自然就是另一个命名空间中的 veth1 了，因为它们互为 peer。

3. 网桥

Linux 可以支持很多不同的端口，这些端口之间当然应该能够通信，如何将这些端口连接起来并实现类似交换机那样的多对多通信呢？这就是网桥的作用了。网桥是一个二层网络设备，可以解析收发的报文，读取目标 MAC 地址的信息，和自己记录的 MAC 表结合，来决策报文的转发端口。为了实现这些功能，网桥会学习源 MAC 地址（二层网桥转发的依据就是 MAC 地址）。在转发报文的时候，网桥只需要向特定的网络接口进行转发，从而避免不必要的网络交互。如果它遇到一个自己从未学习到的地址，就无法知道这个报文应该从哪个网口设备转发，于是只好将报文广播给所有的网络设备端口（报文来源的那个端口除外）。

在实际网络中，网络拓扑不可能永久不变。如果设备移动到另一个端口上，而它没有发送任何数据，那么网桥设备就无法感知到这个变化，结果网桥还是向原来的端口转发数据包，在这种情况下数据就会丢失。所以网桥还要对学习到的 MAC 地址表加上超时时间（默认为 5 分钟）。如果网桥收到了对应端口 MAC 地址回发的包，则重置超时时间，否则过了超时时间后，就认为那个设备已经不在那个端口上了，它就会重新广播发送。

在 Linux 的内部网络栈里面实现的网桥设备，作用和上面的描述相同。过去 Linux 主机一般都只有一个网卡，现在多网卡的机器越来越多，而且还有很多虚拟的设备存在，所以 Linux 的网桥提供了这些设备之间互相转发数据的二层设备。

Linux 内核支持网口的桥接（目前只支持以太网接口）。但是与单纯的交换机不同，交换机只是一个二层设备，对于接收到的报文，要么转发，要么丢弃。运行着 Linux 内核的机器本身就是一台主机，有可能是网络报文的目的地，其收到的报文除了转发和丢弃，还可能被送到网络协议栈的上层（网络层），从而被自己（这台主机本身的协议栈）消化，所以我们既可以看作网桥看作一个二层设备，也可以看作一个三层设备。

1) Linux 网桥的实现

Linux 内核是通过一个虚拟的网桥设备（Net Device）来实现桥接的。这个虚拟设备可以绑定若干个以太网接口设备，从而将它们桥接起来。如图 3.19 所示，这种 Net Device 网桥和普通的设备不同，最明显的一个特性是它还可以有一个 IP 地址。

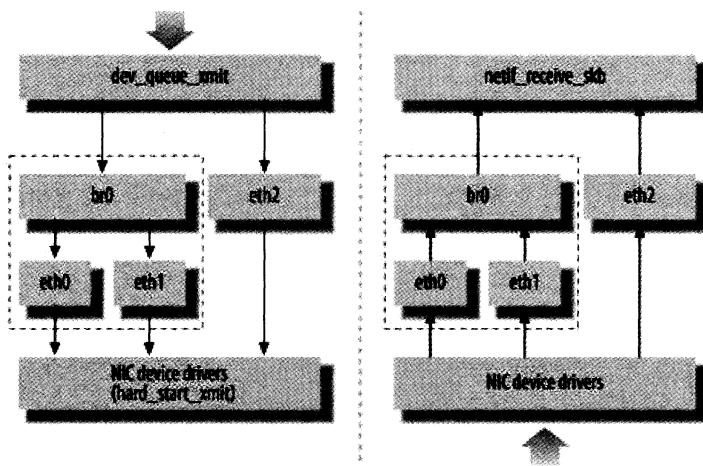


图 3.19 网桥的位置

如图 3.19 所示，网桥设备 br0 绑定了 eth0 和 eth1。对于网络协议栈的上层来说，只看得到 br0。因为桥接是在数据链路层实现的，上层不需要关心桥接的细节，于是协议栈上层需要发送的报文被送到 br0，网桥设备的处理代码判断报文该被转发到 eth0 还是 eth1，或者两者皆转发；反过来，从 eth0 或从 eth1 接收到的报文被提交给网桥的处理代码，在这里会判断报文应该被转发、丢弃还是提交到协议栈上层。

而有时 eth0、eth1 也可能作为报文的源地址或目的地址，直接参与报文的发送与接收，从而绕过网桥。

2) 网桥的常用操作命令

Docker 自动完成了对网桥的创建和维护。为了进一步理解网桥，下面举几个常用的网桥操作例子，对网桥进行手工操作：

```
#brctl addbr xxxx 就是新增一个网桥
```

之后可以增加端口，在 Linux 中，一个端口其实就是一个物理网卡。将物理网卡和网桥连接起来：

```
#brctl addif xxxx ethx
```

网桥的物理网卡作为一个端口，由于在链路层工作，就不再需要 IP 地址了，这样上面的 IP 地址自然失效：

```
#ifconfig ethx 0.0.0.0
```

给网桥配置一个 IP 地址：

```
#ifconfig brxxxx xxx.xxxx.xxxx.xxx
```

这样网桥就有了一个 IP 地址，而连接到上面的网卡就是一个纯链路层设备了。

4. Iptables/Netfilter

我们知道，Linux 网络协议栈非常高效，同时比较复杂。如果我们希望在数据的处理过程中对关心的数据进行一些操作该怎么做呢？Linux 提供了一套机制来为用户实现自定义的数据包处理过程。

在 Linux 网络协议栈中有一组回调函数挂接点，通过这些挂接点挂接的钩子函数可以在 Linux 网络栈处理数据包的过程中对数据包进行一些操作，例如过滤、修改、丢弃等。整个挂接点技术叫作 Netfilter 和 Iptables。

Netfilter 负责在内核中执行各种挂接的规则，运行在内核模式中；而 Iptables 是在用户模式下运行的进程，负责协助维护内核中 Netfilter 的各种规则表。通过二者的配合来实现整个 Linux 网络协议栈中灵活的数据包处理机制。

Netfilter 可以挂接的规则点有 5 个，如图 3.20 中的深色椭圆所示。

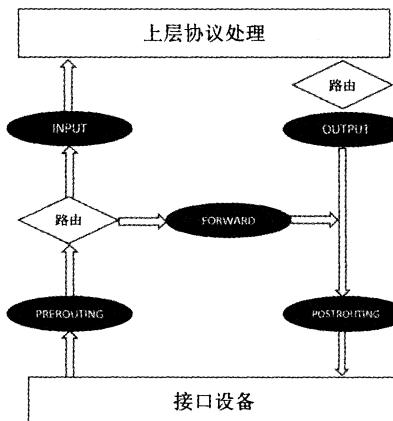


图 3.20 Netfilter 挂接点

1) 规则表 Table

这些挂接点能挂接的规则也分不同的类型（也就是规则表 Table），我们可以在不同类型的 Table 中加入我们的规则。目前主要支持的 Table 类型为：

- ◎ RAW;
- ◎ MANGLE;
- ◎ NAT;
- ◎ FILTER。

上述 4 个 Table（规则链）的优先级是 RAW 最高，FILTER 最低。

在实际应用中，不同的挂接点需要的规则类型通常不同。例如，在 Input 的挂接点上明显不需要 FILTER 过滤规则，因为根据目标地址，已经选择好本机的上层协议栈了，所以无须再挂接 FILTER 过滤规则。目前 Linux 系统支持的不同挂接点能挂接的规则类型如图 3.21 所示。

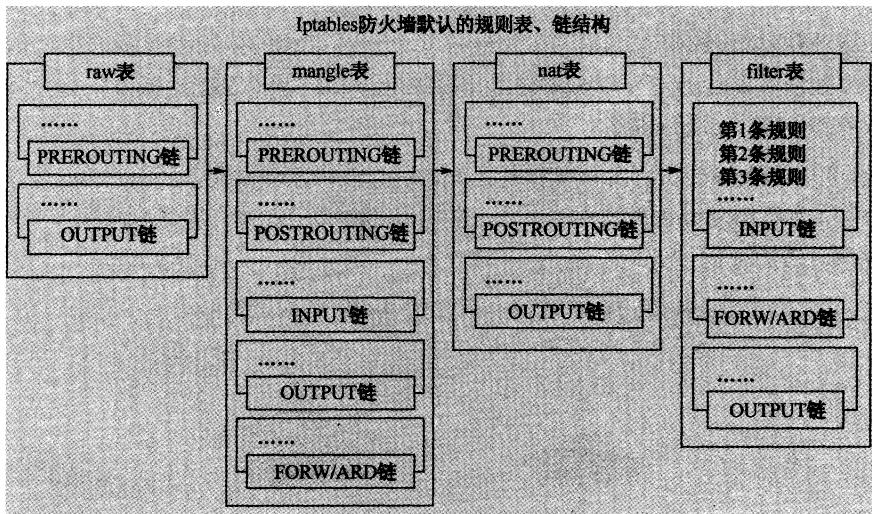


图 3.21 不同表的挂接点

当 Linux 协议栈的数据处理运行到挂接点时，它会依次调用挂接点上所有的挂钩函数，直到数据包的处理结果是明确地接受或者拒绝。

2) 处理规则

每个规则的特性都分为以下几部分：

- ◎ 表类型（准备干什么事情）；
- ◎ 什么挂接点（什么时候起作用）；
- ◎ 匹配的参数是什么（针对什么样的数据包）；
- ◎ 匹配后有什么动作（匹配后具体的操作是什么）。

表类型和什么挂接点在前面已经介绍了，现在我们看看匹配的参数和匹配后的动作。

(1) 匹配的参数

匹配的参数用于对数据包或者 TCP 数据连接的状态进行匹配。当有多个条件存在时，它们一起起作用，来达到只针对某部分数据进行修改的目的。常见的匹配参数有：

- ◎ 流入、流出的网络接口；

- ◎ 来源、目的地址；
- ◎ 协议类型；
- ◎ 来源、目的端口。

(2) 匹配后的动作

一旦有数据匹配上，就会执行相应的动作。动作类型既可以是标准的预定义的几个动作，也可以是自定义的模块注册的动作，或者是一个新的规则链，以便更好地组织一组动作。

3) Iptables 命令

Iptables 命令用于协助用户维护各种规则。我们在使用 Kubernetes、Docker 的过程中，通常都会去查看相关的 Netfilter 配置。这里只介绍如何查看规则表，详细的介绍请参照 Linux 的 Iptables 帮助文档。

查看系统中已有的规则的方法如下。

- ◎ `iptables-save`: 按照命令的方式打印 Iptables 的内容。
- ◎ `Iptables-vnL`: 以另一种格式显示 Netfilter 表的内容。

5. 路由

Linux 系统包含一个完整的路由功能。当 IP 层在处理数据发送或者转发的时候，会使用路由表来决定发往哪里。通常情况下，如果主机与目的主机直接相连，那么主机可以直接发送 IP 报文到目的主机，这个过程比较简单。例如，通过点对点的链接或通过网络共享，如果主机与目的主机没有直接相连，那么主机会将 IP 报文发送给默认的路由器，然后由路由器来决定往哪发送 IP 报文。

路由功能由 IP 层维护的一张路由表来实现。当主机收到数据报文时，它用此表来决策接下来应该做什么操作。当从网络侧接收到数据报文时，IP 层首先会检查报文的 IP 地址是否与主机自身的地址相同。如果数据报文中的 IP 地址是主机自身的地址，那么报文将被发送到传输层相应的协议中去。如果报文中的 IP 地址不是主机自身的地址，并且主机配置了路由功能，那么报文将被转发，否则，报文将被丢弃。

路由表中的数据一般是以条目形式存在的。一个典型的路由表条目通常包含以下主要的条目项。

(1) 目的 IP 地址：此字段表示目标的 IP 地址。这个 IP 地址可以是某台主机的地址，也可以是一个网络地址。如果这个条目包含的是一个主机地址，那么它的主机 ID 将被标记为非零；如果这个条目包含的是一个网络地址，那么它的主机 ID 将被标记为零。

(2) 下一个路由器的 IP 地址：为什么采用“下一个”的说法，是因为下一个路由器并不总是最终的目的路由器，它很可能是一个中间路由器。条目给出下一个路由器的地址用来转发从相应接口接收到的 IP 数据报文。

(3) 标志：这个字段提供了另一组重要信息，例如目的 IP 地址是一个主机地址还是一个网络地址。此外，从标志中可以得知下一个路由器是一个真实路由器还是一个直接相连的接口。

(4) 网络接口规范：为一些数据报文的网络接口规范，该规范将与报文一起被转发。

在通过路由表转发时，如果任何条目的第 1 个字段完全匹配目的 IP 地址（主机）或部分匹配条目的 IP 地址（网络），那么它将指示下一个路由器的 IP 地址。这是一个重要的信息，因为这些信息直接告诉主机（具备路由功能的）数据包应该转发到哪个“下一个路由器”去。而条目中的所有其他字段将提供更多的辅助信息来为路由转发做决定。

如果没有找到一个完全匹配的 IP，那么就接着搜索相匹配的网络 ID。如果找到，那么该数据报文会被转发到指定的路由器上。可以看出，网络上的所有主机都通过这个路由表中的单个（这个）条目进行管理。

如果上述两个条件都不匹配，那么该数据报文将被转发到一个默认路由器上。

如果上述步骤失败，默认路由器也不存在，那么该数据报文最终无法被转发。任何无法投递的数据报文都将产生一个 ICMP 主机不可达或 ICMP 网络不可达的错误，并将此错误返回给生成此数据报文的应用程序。

1) 路由表的创建

Linux 的路由表至少包括两个表（当启用策略路由的时候，还会有其他表）：一个是 LOCAL，另一个是 MAIN。在 LOCAL 表中会包含所有的本地设备地址。LOCAL 路由表是在配置网络设备地址时自动创建的。LOCAL 表用于供 Linux 协议栈识别本地地址，以及进行本地各个不同网络接口之间的数据转发。

可以通过下面的命令查看 LOCAL 表的内容：

```
# ip route show table local type local
10.1.1.0 dev flannel0 proto kernel scope host src 10.1.1.0
127.0.0.0/8 dev lo proto kernel scope host src 127.0.0.1
127.0.0.1 dev lo proto kernel scope host src 127.0.0.1
172.17.42.1 dev docker proto kernel scope host src 172.17.42.1
192.168.1.128 dev eno1677736 proto kernel scope host src 192.168.1.128
```

MAIN 表用于各类网络 IP 地址的转发。它的建立既可以使用静态配置生成，也可以使用动态路由发现协议生成。动态路由发现协议一般使用组播功能来通过发送路由发现数据，动态地交换和获取网络的路由信息，并更新到路由表中。

Linux下支持路由发现协议的开源软件有许多，常用的有Quagga、Zebra等。第4章会介绍使用Quagga动态容器路由发现的机制来实现Kubernetes的网络组网。

2) 路由表的查看

我们可以使用`ip route list`命令查看当前的路由表。

```
# ip route list
192.168.6.0/24 dev eno1677736 proto kernel scope link src 192.168.6.140
metric 1
```

在上面的例子代码中，只有一个子网的路由，源地址是192.168.6.140（本机），目标地址是192.168.6.0/24网段的数据，都将通过eth0接口设备发送出去。

`Netstat-rn`是另一个查看路由表的工具：

```
# netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask        Flags   MSS Window irtt Iface
0.0.0.0         192.168.6.2   0.0.0.0       UG        0 0          0 eth0
192.168.6.0     0.0.0.0       255.255.255.0 U          0 0          0 eth0
```

在它显示的信息中，如果标志是U，则说明是可达路由；如果标志是G，则说明这个网络接口连接的是网关，否则说明是直连主机。

3.7.3 Docker的网络实现

标准的Docker支持以下4类网络模式。

- ◎ host模式：使用`--net=host`指定。
- ◎ container模式：使用`--net=container:NAME_or_ID`指定。
- ◎ none模式：使用`--net=none`指定。
- ◎ bridge模式：使用`--net=bridge`指定，为默认设置。

在Kubernetes管理模式下，通常只会使用bridge模式，所以本节只介绍bridge模式下Docker是如何支持网络的。

在bridge模式下，Docker Daemon第一次启动时会创建一个虚拟的网桥，默认的名字是`docker0`，然后按照RPC1918的模型，在私有网络空间中给这个网桥分配一个子网。针对由Docker创建出来的每一个容器，都会创建一个虚拟的以太网设备（Veth设备对），其中一端关联到网桥上，另一端使用Linux的网络命名空间技术，映射到容器内的`eth0`设备，然后从网桥的地址段内给`eth0`接口分配一个IP地址。

如图 3.22 所示就是 Docker 的默认桥接网络模型。

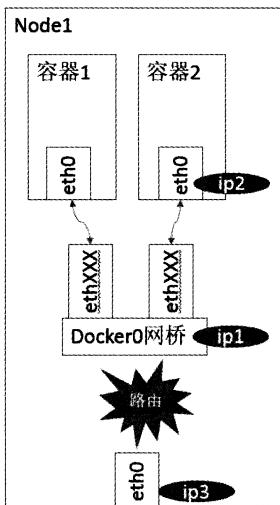


图 3.22 默认的 Docker 网络桥接模型

其中 ip1 是网桥的 IP 地址，Docker Daemon 会在几个备选地址段里给它选一个，通常是 172 开头的一个地址。这个地址和主机的 IP 地址是不重叠的。ip2 是 Docker 在启动容器的时候，在这个地址段随机选择的一个没有使用的 IP 地址，Docker 占用它并分配给了被启动的容器。相应的 MAC 地址也根据这个 IP 地址，在 02:42:ac:11:00:00 和 02:42:ac:11:ff:ff 的范围内生成，这样做可以确保不会有 ARP 的冲突。

启动后，Docker 还将 Veth 对的名字映射到了 eth0 网络接口。ip3 就是主机的网卡地址。

在一般情况下，ip1、ip2 和 ip3 是不同的 IP 段，所以在默认不做任何特殊配置的情况下，在外部是看不到 ip1 和 ip2 的。

这样做的结果就是，同一台机器内的容器之间可以相互通信。不同主机上的容器不能够相互通信。实际上它们甚至有可能会在相同的网络地址范围内（不同的主机上的 docker0 的地址段可能是一样的）。

为了让它们跨节点互相通信，就必须在主机的地址上分配端口，然后通过这个端口路由或代理到容器上。这种做法显然意味着一定要在容器之间小心谨慎地协调好端口的分配，或者使用动态端口的分配技术。在不同应用之间协调好端口分配是十分困难的事情，特别是集群水平扩展的时候。而动态的端口分配也会带来高度复杂性，例如：每个应用程序都只能将端口看作一个符号（因为是动态分配的，无法提前设置）。而且 API Server 也要在分配完后，将动态端口插入到配置的合适位置。另外，服务也必须能互相之间找到对方等。这些都是 Docker 的网络模

型在跨主机访问时面临的问题。

1) 查看 Docker 启动后的系统情况

我们已经知道，Docker 网络在 bridge 模式下 Docker Daemon 启动时创建 docker0 网桥，并在网桥使用的网段为容器分配 IP。让我们看看实际的操作。

在刚刚启动 Docker Daemon 并且还没有启动任何容器的时候，网络协议栈的配置情况如下：

```
# systemctl start docker
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eno1677736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP qlen 1000
    link/ether 00:0c:29:14:3d:80 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.133/24 brd 192.168.1.255 scope global eno1677736
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe14:3d80/64 scope link
        valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:6e:af:0e:c3 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/24 scope global docker0
        valid_lft forever preferred_lft forever

# iptables-save
# Generated by iptables-save v1.4.21 on Thu Sep 24 17:11:04 2015
*nat
:PREROUTING ACCEPT [7:878]
:INPUT ACCEPT [7:878]
:OUTPUT ACCEPT [3:536]
:POSTROUTING ACCEPT [3:536]
:DOCKER - [0:0]
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
COMMIT
# Completed on Thu Sep 24 17:11:04 2015
# Generated by iptables-save v1.4.21 on Thu Sep 24 17:11:04 2015
*filter
:INPUT ACCEPT [133:11362]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [37:5000]
:DOCKER - [0:0]
```

```

-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
COMMIT
# Completed on Thu Sep 24 17:11:04 2015

```

可以看到，Docker 创建了 docker0 网桥，并添加了 Iptables 规则。docker0 网桥和 Iptables 规则都处于 root 命名空间中。通过解读这些规则，我们发现，在还没有启动任何容器时，如果启动了 Docker Daemon，那么它就已经做好了通信的准备。对这些规则的说明如下。

(1) 在 NAT 表中有 3 条记录，前两条匹配生效后，都会继续执行 DOCKER 链，而此时 DOCKER 链为空，所以前两条只是做了个框架，并没有实际效果。

(2) NAT 表第 3 条的含义是，若本地发出的数据包不是发往 docker0 的，即是发往主机之外的设备的，都需要进行动态地址修改 (MASQUERADE)，将源地址从容器的地址 (172 段) 修改为宿主机网卡的 IP 地址，之后就可以发送给外面的网络了。

(3) 在 FILTER 表中，第 1 条也是一个框架，因为后继的 DOCKER 链是空的。

(4) 在 FILTER 表中，第 3 条是说，docker0 发出的包，如果需要 Forward 到非 docker0 的本地 IP 地址的设备，则是允许的，这样，docker0 设备的包就可以根据路由规则中转到宿主机的网卡设备，从而访问外面的网络。

(5) FILTER 表中，第 4 条是说，docker0 的包还可以中转给 docker0 本身，即连接在 docker0 网桥上的不同容器之间的通信也是允许的。

(6) FILTER 表中，第 2 条是说，如果接收到的数据包属于以前已经建立好的连接，那么允许直接通过。这样接收到的数据包自然又走回 docker0，并中转到相应的容器。

除了这些 Netfilter 的设置，Linux 的 ip_forward 功能也被 Docker Daemon 打开了：

```
# cat /proc/sys/net/ipv4/ip_forward
1
```

另外，我们还可以看到刚刚启动 Docker 后的 Route 表，和启动前没有什么不同：

```
# ip route
default via 192.168.1.2 dev eno1677736 proto static metric 100
172.17.0.0/16 dev docker proto kernel scope link src 172.17.42.1
192.168.1.0/24 dev eno1677736 proto kernel scope link src 192.168.1.132
192.168.1.0/24 dev eno1677736 proto kernel scope link src 192.168.1.132
metric 100
```

2) 查看容器启动后的情况（容器无端口映射）

刚才我们看了 Docker 服务启动后的网络情况。现在，我们启动一个 Registry 容器后（不使

用任何端口镜像参数), 看一下网络堆栈部分相关的变化:

```

docker run --name register -d registry
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP qlen 1000
    link/ether 00:0c:29:c8:12:5f brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.132/24 brd 192.168.1.255 scope global eno16777736
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fec8:125f/64 scope link
        valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
    link/ether 02:42:72:79:b8:88 brd ff:ff:ff:ff:ff:ff
    inet 172.17.42.1/24 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:7aff:fe79:b888/64 scope link
        valid_lft forever preferred_lft forever
13: veth2dc8bbd: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
docker0 state UP
    link/ether be:d9:19:42:46:18 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::bcd9:19ff:fe42:4618/64 scope link
        valid_lft forever preferred_lft forever

# iptables-save
# Generated by iptables-save v1.4.21 on Thu Sep 24 18:21:04 2015
*nat
:PREROUTING ACCEPT [14:1730]
:INPUT ACCEPT [14:1730]
:OUTPUT ACCEPT [59:4918]
:POSTROUTING ACCEPT [59:4918]
:DOCKER - [0:0]
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
COMMIT
# Completed on Thu Sep 24 18:21:04 2015
# Generated by iptables-save v1.4.21 on Thu Sep 24 18:21:04 2015
*filter
:INPUT ACCEPT [2383:211572]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [2004:242872]
```

```
:DOCKER - [0:0]
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
COMMIT
# Completed on Thu Sep 24 18:21:04 2015

# ip route
default via 192.168.1.2 dev eno1677736 proto static metric 100
172.17.0.0/16 dev docker proto kernel scope link src 172.17.42.1
192.168.1.0/24 dev eno1677736 proto kernel scope link src 192.168.1.132
192.168.1.0/24 dev eno1677736 proto kernel scope link src 192.168.1.132
metric 100
```

可以看到如下情况。

(1) 宿主机器上的 Netfilter 和路由表都没有变化，说明在不进行端口映射时，Docker 的默认网络是没有特殊处理的。相关的 NAT 和 FILTER 两个 Netfilter 链还是空的。

(2) 宿主机上的 Veth 对已经建立，并连接到了容器内。

我们再次进入刚刚启动的容器内，看看网络栈是什么情况。容器内部的 IP 地址和路由如下：

```
# docker exec -ti 24981a750a1a bash
[root@24981a750a1a /]# ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0 proto kernel scope link src 172.17.0.10
[root@24981a750a1a /]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
22: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:0a brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.10/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:a/64 scope link
        valid_lft forever preferred_lft forever
```

我们可以看到，默认停止的回环设备 lo 已经被启动，外面宿主机连接进来的 Veth 设备也被命名成了 eth0，并且已经配置了地址 172.17.0.10。

路由信息表包含一条到 docker0 的子网路由和一条到 docker0 的默认路由。

3) 查看容器启动后的情况（容器有端口映射）

下面，我们用带端口映射的命令启动 registry：

```
docker run --name register -d -p 1180:5000 registry
```

在启动后查看 Iptables 的变化。

```
# iptables-save
# Generated by iptables-save v1.4.21 on Thu Sep 24 18:45:13 2015
*nat
:PREROUTING ACCEPT [2:236]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
:DOCKER - [0:0]
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A POSTROUTING -s 172.17.0.19/32 -d 172.17.0.19/32 -p tcp -m tcp --dport 5000
-j MASQUERADE
-A DOCKER ! -i docker0 -p tcp -m tcp --dport 1180 -j DNAT --to-destination
172.17.0.19:5000
COMMIT
# Completed on Thu Sep 24 18:45:13 2015
# Generated by iptables-save v1.4.21 on Thu Sep 24 18:45:13 2015
*filter
:INPUT ACCEPT [54:4464]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [41:5576]
:DOCKER - [0:0]
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER -d 172.17.0.19/32 ! -i docker0 -o docker0 -p tcp -m tcp --dport 5000
-j ACCEPT
COMMIT
# Completed on Thu Sep 24 18:45:13 2015
```

从新增的规则可以看出，Docker 服务在 NAT 和 FILTER 两个表内添加的两个 DOCKER 子链都是给端口映射用的。例如本例中我们需要把外面宿主机的 1180 端口映射到容器的 5000 端口。通过前面的分析我们知道，无论是宿主机接收到的还是宿主机本地协议栈发出的，目标地址是本地 IP 地址的包都会经过 NAT 表中的 DOCKER 子链。Docker 为每一个端口映射都在这个链上增加了到实际容器目标地址和目标端口的转换。

经过这个 DNAT 的规则修改后的 IP 包，会重新经过路由模块的判断进行转发。由于目标地

址和端口已经是容器的地址和端口，所以数据自然就送到了 docker0 上，从而送到对应的容器内部。

当然在 Forward 时，也需要在 Docker 子链中添加一条规则，如果目标端口和地址是指定容器的数据，则允许通过。

在 Docker 按照端口映射的方式启动容器时，主要的不同就是上述 Iptables 部分。而容器内部的路由和网络设备，都和不做端口映射时一样，没有任何变化。

4) Docker 的网络局限

我们从 Docker 对 Linux 网络协议栈的操作可以看到，Docker 一开始没有考虑到多主机互联的网络解决方案。

Docker 一直以来的理念都是“简单为美”，几乎所有尝试 Docker 的人，都被它“用法简单，功能强大”的特性所吸引，这也是 Docker 迅速走红的一个原因。

我们都知道，虚拟化技术中最为复杂的部分就是虚拟化网络技术，即使是单纯的物理网络部分，也是一个门槛很高的技能领域，通常只被少数网络工程师所掌握，所以我们可以理解，结合了物理网络的虚拟网络技术会有多难了。在 Docker 之前，所有接触过 OpenStack 的人的心里都有一个难以释怀的阴影，那就是它的网络问题，于是，Docker 明智地避开这个“雷区”，让其他专业人员去用现有的虚拟化网络技术解决 Docker 主机的互联问题，以免让用户觉得 Docker 太难了，从而放弃学习和使用 Docker。

Docker 成名以后，重新开始重视网络解决方案，收购了一家 Docker 网络解决方案公司——Socketplane，原因在于这家公司的产品广受好评，但有趣的是 Socketplane 的方案就是以 Open vSwitch 为核心的，其还为 Open vSwitch 提供了 Docker 镜像，以方便部署程序。之后，Docker 开启了一个“宏伟”的虚拟化网络解决方案——Libnetwork，如图 3.23 所示是其概念图。

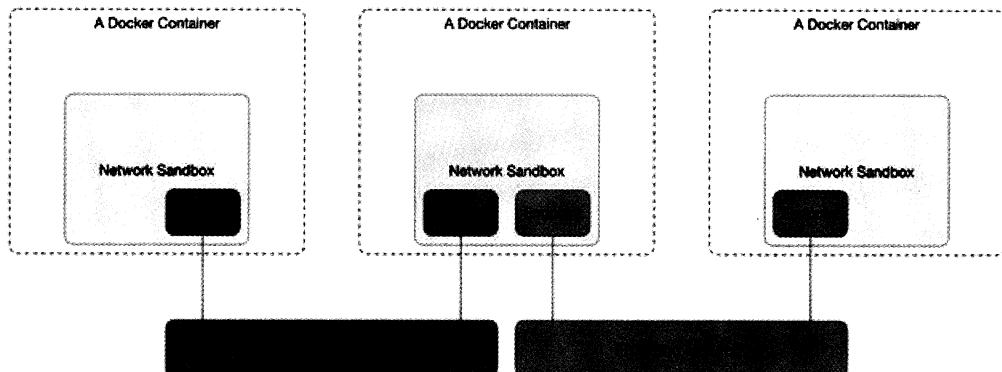


图 3.23 Libnetwork 概念图

这个概念图没有了IP，也没有了路由，已经颠覆了我们的网络常识了，对于不怎么懂网络的大多数人来说，它的确很有诱惑力，未来是否会对虚拟化网络的模型产生深远冲击我们还不得而知，但当前，它仅仅是Docker官方的一次“尝试”。

针对目前Docker的网络实现，Docker使用的Libnetwork组件只是将Docker平台中的网络子系统模块化为一个独立库的简单尝试，离成熟和完善还有一段距离。

所以，直到现在，仍然没有来自Docker官方的可以用于生产实践中的多主机网络解决方案。

3.7.4 Kubernetes的网络实现

在实际的业务场景中，业务组件之间的关系十分复杂，特别是微服务概念的推进，应用部署的粒度更加细小和灵活。为了支持业务应用组件的通信联系，Kubernetes网络的设计主要致力于解决以下场景。

- (1) 容器到容器之间的直接通信。
- (2) 抽象的Pod到Pod之间的通信。
- (3) Pod到Service之间的通信。
- (4) 集群外部与内部组件之间的通信。

其中第3条、第4条我们在之前的章节里都讲述过，本节中我们对更为基础的第1条与第2条进行深入分析和讲解。

1. 容器到容器的通信

在同一个Pod内的容器（Pod内的容器是不会跨宿主机的）共享同一个网络命名空间，共享同一个Linux协议栈。所以对于网络的各类操作，就和它们在同一台机器上一样，它们甚至可以用localhost地址访问彼此的端口。

这么做的结果是简单、安全和高效，也能减少将已经存在的程序从物理机或者虚拟机移植到容器下运行的难度。在容器技术出来之前，其实大家早就积累了如何在一台机器上运行一组应用程序的经验，例如，如何让端口不冲突，以及如何让客户端发现它们等。

我们来看一下Kubernetes是如何利用Docker的网络模型的。

图3.24中的阴影部分就是在Node上运行着的一个Pod实例。在我们的例子中，容器就是图3.24中的容器1和容器2。容器1和容器2共享了一个网络的命名空间，共享一个命名空间的结果就是它们好像在一台机器上运行似的，它们打开的端口不会有冲突，可以直接使用Linux的本地IPC进行通信（例如消息队列或者管道）。其实这和传统的一组普通程序运行的环境是完

全一样的，传统的程序不需要针对网络做特别的修改就可以移植了。它们之间的互相访问只需要使用 `localhost` 就可以。例如，如果容器 2 运行的是 MySQL，那么容器 1 使用 `localhost:3306` 就能直接访问这个运行在容器 2 上的 MySQL 了。

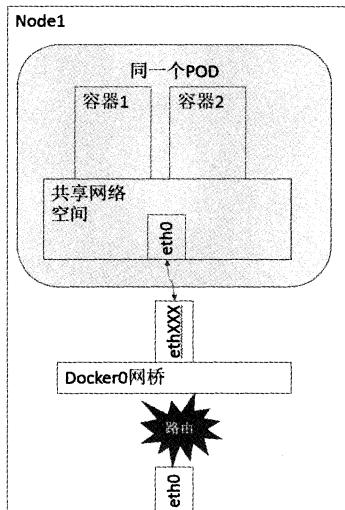


图 3.24 Kubernetes 的 Pod 网络模型

2. Pod 之间的通信

我们看了同一个 Pod 内的容器之间的通信情况，再看看 Pod 之间的通信情况。

每一个 Pod 都有一个真实的全局 IP 地址，同一个 Node 内的不同 Pod 之间可以直接采用对方 Pod 的 IP 地址通信，而且不需要使用其他发现机制，例如 DNS、Consul 或者 etcd。

Pod 容器既有可能在同一个 Node 上运行，也有可能在不同的 Node 上运行，所以通信也分为两类：同一个 Node 内的 Pod 之间的通信和不同 Node 上的 Pod 之间的通信。

1) 同一个 Node 内的 Pod 之间的通信

我们看一下同一个 Node 上的两个 Pod 之间的关系，如图 3.25 所示。

可以看出，Pod1 和 Pod2 都是通过 Veth 连接在同一个 docker0 网桥上的，它们的 IP 地址 IP1、IP2 都是从 docker0 的网段上动态获取的，它们和网桥本身的 IP3 是同一个网段的。

另外，在 Pod1、Pod2 的 Linux 协议栈上，默认路由都是 docker0 的地址，也就是说所有非本地地址的网络数据，都会被默认发送到 docker0 网桥上，由 docker0 网桥直接中转。

综上所述，由于它们都关联在同一个 docker0 网桥上，地址段相同，所以它们之间是能直接通信的。

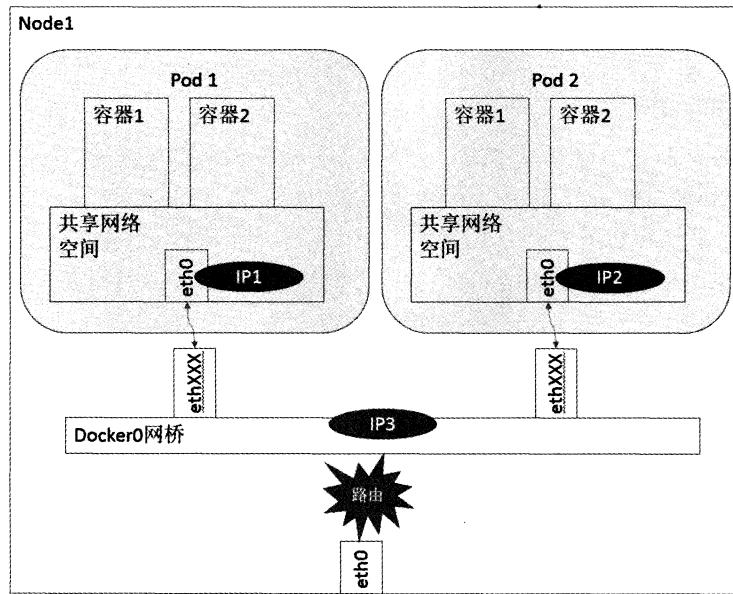


图 3.25 同一个 Node 内的 Pod 关系

2) 不同 Node 上的 Pod 之间的通信

Pod 的地址是与 docker0 在同一个网段内的，我们知道 docker0 网段与宿主机网卡是两个完全不同的 IP 网段，并且不同 Node 之间的通信只能通过宿主机的物理网卡进行，因此要想实现位于不同 Node 上的 Pod 容器之间的通信，就必须想办法通过主机的这个 IP 地址来进行寻址和通信。

另一方面，这些动态分配且藏在 docker0 之后的所谓“私有”IP 地址也是可以找到的。Kubernetes 会记录所有正在运行 Pod 的 IP 分配信息，并将这些信息保存在 etcd 中（作为 Service 的 Endpoint）。这些私有 IP 信息对于 Pod 到 Pod 的通信也是十分重要的，因为我们的网络模型要求 Pod 到 Pod 使用私有 IP 进行通信。所以首先要知道这些 IP 是什么。

之前提到，Kubernetes 的网络对 Pod 的地址是平面的和直达的，所以这些 Pod 的 IP 规划也很重要，不能有冲突。只要没有冲突，我们就可以想办法在整个 Kubernetes 的集群中找到它。

综上所述，要想支持不同 Node 上的 Pod 之间的通信，就要达到两个条件：

- (1) 在整个 Kubernetes 集群中对 Pod 的 IP 分配进行规划，不能有冲突；
- (2) 找到一种办法，将 Pod 的 IP 和所在 Node 的 IP 关联起来，通过这个关联让 Pod 可以互相访问。

根据条件 1 的要求，我们需要在部署 Kubernetes 的时候，对 docker0 的 IP 地址进行规划，

保证每一个 Node 上的 docker0 地址没有冲突。我们可以在规划后手工配置到每个 Node 上，或者做一个分配规则，由安装的程序自己去分配占用。例如 Kubernetes 的网络增强开源软件 Flannel 就能够管理资源池的分配。

根据条件 2 的要求，Pod 中的数据在发出时，需要有一个机制能够知道对方 Pod 的 IP 地址挂接在哪个具体的 Node 上。也就是说先要找到 Node 对应宿主机的 IP 地址，将数据发送到这个宿主机的网卡上，然后在宿主机上将相应的数据转到具体的 docker0 上。一旦数据到达宿主机 Node，则那个 Node 内部的 docker0 便知道如何将数据发送到 Pod。如图 3.26 所示。

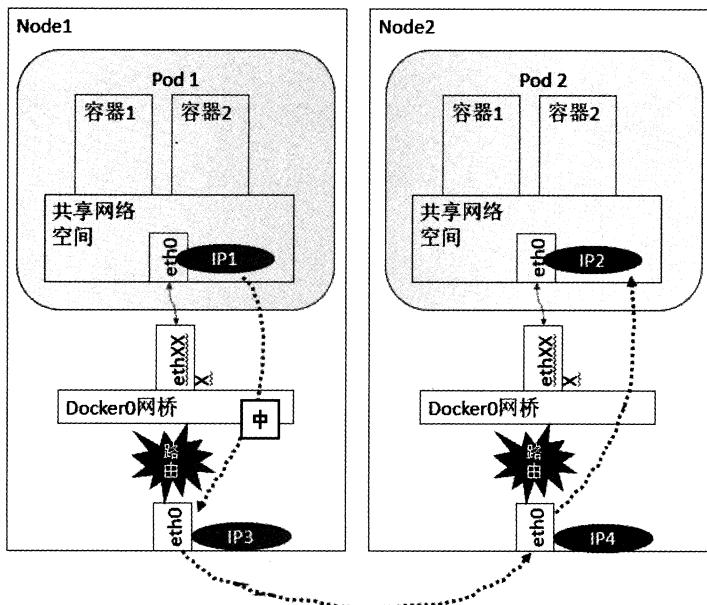


图 3.26 跨 Node 的 Pod 通信

在图 3.26 中，IP1 对应的是 Pod1，IP2 对应的是 Pod2。Pod1 在访问 Pod2 时，首先要将数据从源 Node 的 eth0 发送出去，找到并到达 Node2 的 eth0。也就是说先要从 IP3 到 IP4，之后才是 IP4 到 IP2 的递送。

在谷歌的 GCE 环境下，Pod 的 IP 管理（类似 docker0）、分配及它们之间的路由打通都是由 GCE 完成的。Kubernetes 作为主要在 GCE 上面运行的框架，它的设计是假设底层已经具备这些条件，所以它分配完地址并将地址记录下来就完成了它的工作。在实际的 GCE 环境中，GCE 的网络组件会读取这些信息，实现具体的网络打通。

而在实际的生产中，因为安全、费用、合规等种种原因，Kubernetes 的客户不可能全部使用谷歌的 GCE 环境，所以在实际的私有云环境中，除了部署 Kubernetes 和 Docker，还需要额

外的网络配置，甚至通过一些软件来实现 Kubernetes 对网络的要求。做到这些后，Pod 和 Pod 之间才能无差别地透明通信。

为了达到这个目的，开源界有不少应用来增强 Kubernetes、Docker 的网络，在后面的章节里会介绍几个常用的组件和它们的组网原理。

3.7.5 开源的网络组件

Kubernetes 的网络模型假定了所有 Pod 都在一个可以接连通的扁平的网络空间中。这在 GCE 里面是现成的网络模型，Kubernetes 假定这个网络已经存在。而在私有云里搭建 Kubernetes 集群，就不能假定这种网络已经存在了。我们需要自己实现这个网络假设，将不同节点上的 Docker 容器之间的互相访问先打通，然后运行 Kubernetes。

目前已经多个开源组件支持这个网络模型。这里介绍几个常见的模型，分别是 Flannel、Open vSwitch 及直接路由的方式。

1. Flannel

Flannel 之所以可以搭建 Kubernetes 依赖的底层网络，是因为它能实现以下两点。

(1) 它能协助 Kubernetes，给每一个 Node 上的 Docker 容器分配互相不冲突的 IP 地址。

(2) 它能在这些 IP 地址之间建立一个覆盖网络 (Overlay Network)，通过这个覆盖网络，将数据包原封不动地传递到目标容器内。

通过图 3.27 来看看 Flannel 是如何实现这两点的。

可以看到，Flannel 首先创建了一个名为 flannel0 的网桥，而且这个网桥的一端连接 docker0 网桥，另一端连接一个叫作 flanneld 的服务进程。

flanneld 进程并不简单，它首先上连 etcd，利用 etcd 来管理可分配的 IP 地址段资源，同时监控 etcd 中每个 Pod 的实际地址，并在内存中建立了一个 Pod 节点路由表；然后下连 docker0 和物理网络，使用内存中的 Pod 节点路由表，将 docker0 发给它的数据包包装起来，利用物理网络的连接将数据包投递到目标 flanneld 上，从而完成 Pod 到 Pod 之间的直接的地址通信。

Flannel 之间的底层通信协议的可选余地很多，有 UDP、VxLan、AWS VPC 等多种方式，只要能通到对端的 Flannel 就可以了。源 flanneld 加包，目标 flanneld 解包，最终 docker0 看到的就是原始的数据，非常透明，根本感觉不到中间 Flannel 的存在。常用的是 UDP。

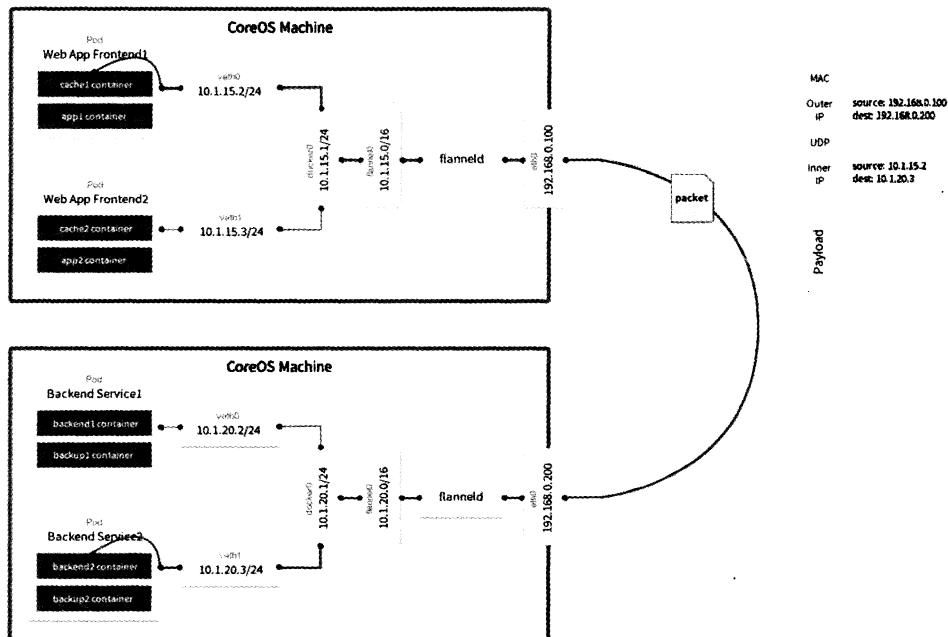


图 3.27 Flannel 架构图

我们看一下 Flannel 是如何做到让为不同 Node 上的 Pod 分配的 IP 不产生冲突的。其实想到 Flannel 使用了集中的 etcd 存储就很容易理解了。它每次分配的地址段都在同一个公共区域获取，这样大家自然能够互相协调，不产生冲突了。而且在 Flannel 分配好地址段后，后面的事情是由 Docker 完成的，Flannel 通过修改 Docker 的启动参数将分配给它的地址段传递进去。

```
--bip=172.17.18.1/24
```

通过这些操作，Flannel 就控制了每个 Node 上的 docker0 地址段的地址，也就保障了所有 Pod 的 IP 地址在同一个水平网络中且不产生冲突了。

Flannel 完美地实现了对 Kubernetes 网络的支持，但是它引入了多个网络组件，在网络通信时需要转到 flannel0 网络接口，再转到用户态的 flanneld 程序，到对端后还需要走这个过程的反过程，所以也会引入一些网络的时延损耗。

另外，Flannel 模型默认使用了 UDP 作为底层传输协议，UDP 本身是非可靠协议，虽然两端的 TCP 实现了可靠传输，但在大流量、高并发应用场景下还需要反复测试，确保没有问题。

2. Open vSwitch

在了解了 Flannel 后，我们再看看 Open vSwitch 是怎么解决上述两个问题的。

Open vSwitch 是一个开源的虚拟交换机软件，有点儿像 Linux 中的 bridge，但是功能要复

杂得多。Open vSwitch 的网桥可以直接建立多种通信通道（隧道），例如 Open vSwitch with GRE/VxLAN。这些通道的建立可以很容易地通过 OVS 的配置命令实现。在 Kubernetes、Docker 场景下，我们主要是建立 L3 到 L3 的隧道。举一个例子来看看 Open vSwitch with GRE/VxLAN 的网络架构，如图 3.28 所示。

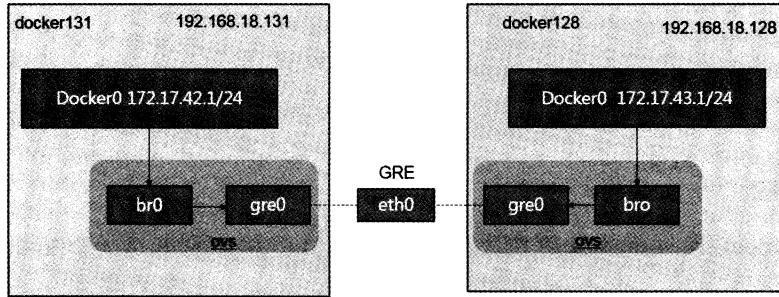


图 3.28 OVS with GRE 原理图

首先，为了避免 Docker 创建的 docker0 地址产生冲突（因为 Docker Daemon 启动且给 docker0 选择子网地址时只有几个备选列表，很容易产生冲突），我们可以将 docker0 网桥删除，手动建立一个 Linux 网桥，然后手动给这个网桥配置 IP 地址范围。

其次，建立 Open vSwitch 的网桥 ovs，然后使用 ovs-vsctl 命令给 ovs 网桥增加 gre 端口，添加 gre 端口时要将目标连接的 NodeIP 地址设置为对端的 IP 地址。对每一个对端 IP 地址都需要这么操作（对于大型集群网络，这可是个体力活，要做自动化脚本来完成）。

最后将 ovs 的网桥作为网络接口，加入 Docker 的网桥上（docker0 或者自己手工建立的新网桥）。

重启 ovs 网桥和 Docker 的网桥，并添加一个 Docker 的地址段到 Docker 网桥的路由规则项，就可以将两个容器的网络连接起来了。

1) 网络通信过程

当容器内的应用访问另一个容器的地址时，数据包会通过容器内的默认路由发送给 docker0 网桥。ovs 的网桥是作为 docker0 网桥的端口存在的，它会将数据发送给 ovs 网桥。ovs 网络已经通过配置建立了和其他 ovs 网桥的 GRE/VxLAN 隧道，自然能将数据送达对端的 Node，并送往 docker0 及 Pod。

通过新增的路由项，使得 Node 节点本身的应用的数据也路由到 docker0 网桥上，和刚才的通信过程一样，自然也可以访问其他 Node 上的 Pod。

2) OVS with GRE/VxLAN 组网方式的特点

OVS 的优势是，作为开源虚拟交换机软件，它相对比较成熟和稳定，而且支持各类网络隧

道协议，经过了 OpenStack 等项目的考验。

另一方面，在前面介绍 Flannel 的时候可知 Flannel 除了支持建立覆盖网络(Overlay Network)，保证 Pod 到 Pod 的无缝通信，还和 Kubernetes、Docker 架构体系结合紧密。Flannel 能够感知 Kubernetes 的 Service，动态维护自己的路由表，还通过 etcd 来协助 Docker 对整个 Kubernetes 集群中 docker0 的子网地址分配。而我们在使用 OVS 的时候，很多事情就需要手工完成了。

无论是 OVS 还是 Flannel，通过覆盖网络提供的 Pod 到 Pod 通信都会引入一些额外的通信开销，如果是对网络依赖特别重的应用，则需要评估对业务的影响。

3. 直接路由

我们知道，docker0 网桥上的 IP 地址在 Node 网络上是看不到的。从一个 Node 到另一个 Node 内的 docker0 是不通的。因为它不知道某个 IP 地址在哪里。如果能够让这些机器知道对端 docker0 地址在哪里，就可以让这些 docker0 互相通信了。这样所有 Node 上运行的 Pod 就可以互相通信了。

我们可以通过部署 MultiLayer Switch (MLS) 来实现这一点，在 MLS 中配置每个 docker0 子网地址到 Node 地址的路由项，通过 MLS 将 docker0 的 IP 寻址定向到对应的 Node 节点上。

另外，我们还可以将这些 docker0 和 Node 的匹配关系配置在 Linux 操作系统的路由项中，这样通信发起的 Node 能够根据这些路由信息直接找到目标 Pod 所在的 Node，将数据传输过去。如图 3.29 所示。

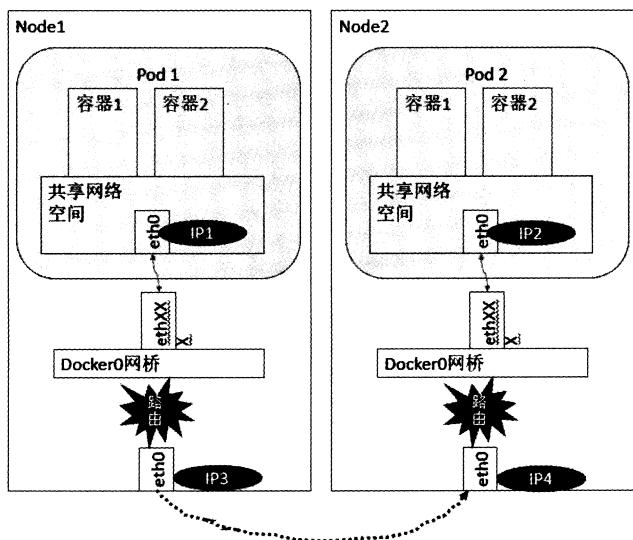


图 3.29 直接路由 Pod 到 Pod 通信

我们在每个 Node 的路由表中增加对方所有 docker0 的路由项。

例如 Pod1 所在 docker0 网桥的 IP 子网是 10.1.10.0, Node 的地址为 192.168.1.128; 而 Pod2 所在 docker0 网桥的 IP 子网是 10.1.20.0, Node 的地址为 192.168.1.129。

在 Node1 上用 route add 命令增加一条到 Node2 上 docker0 的静态路由规则:

```
route add -net 10.1.20.0 netmask 255.255.255.0 gw 192.168.1.129
```

同样, 在 Node2 上增加一条到 Node1 上 docker0 的静态路由规则:

```
route add -net 10.1.10.0 netmask 255.255.255.0 gw 192.168.1.128
```

这样两个 Node 之间的 Pod 就可以互相通信了, 因为它们发出的数据包经过本地 Linux 的路由规则, 能将数据送到对端的 Node。

在大规模集群中, 在每个 Node 上都需要配置到其他 docker0/Node 的路由项, 会带来很大的工作量; 并且在新增机器时, 对所有 Node 都需要修改配置; 重启机器时, 如果 docker0 的地址有变化, 则也需要修改所有 Node 的配置, 这显然是非常复杂的。

为了管理这些动态变化的 docker0 地址, 动态地让其他 Node 都感知到它, 还可以使用动态路由发现协议来同步这些变化。运行动态路由发现协议代理的 Node, 会将本机 LOCAL 路由表的 IP 地址通过组播协议发布出去, 同时监听其他 Node 的组播包。通过这样的信息交换, Node 上的路由规则都能够相互学习到。当然, 路由发现协议本身还是很复杂的, 感兴趣的话你可以查阅相关的规范。在实现这些动态路由发现协议的开源软件中, 常用的有 Quagga、Zebra 等。下面简单介绍直接路由的操作过程。

(1) 首先手工分配 Docker bridge 的地址, 保证它们在不同的网段是不重叠的。建议最好不用 Docker Daemon 自动创建的 docker0 (因为我们不需要它的自动管理功能), 而是单独建立一个 bridge, 给它配置规划好的 IP 地址, 然后使用--bridge=XX 来指定网桥。

(2) 然后在每一个节点上运行 Quagga。

完成这些操作后, 我们很快就能得到一个 Pod 和 Pod 直接互相访问的环境了。由于路由发现能够被网络上的所有设备接收, 所以如果网络上的路由器也能打开 RIP 协议选项, 则能够学习到这些路由信息。通过这些路由器, 我们甚至可以在非 Node 节点上使用 Pod 的 IP 地址直接访问 Node 上的 Pod。

当然, 聪明的你还会新的疑问: 这样做的话, 由于每一个 Pod 的地址都会被路由发现协议广播出去, 会不会存在路由表过大的情况? 实际上, 路由表通常都会有高速缓存, 查找速度会很快, 不会对性能产生太大的影响。当然, 如果你的集群容量在数千台 Node 以上, 则仍然需要测试和评估路由表的效率问题。

3.7.6 网络实战

Docker 给我们带来了不同的网络模式，而 Kubernetes 也以一种不同的方式来解决这些网络模式的挑战，但是其方式有些不太好理解，特别是对于刚开始接触 Kubernetes 的网络的开发者。我们在前面学习了 Kubernetes、Docker 的理论，本节将通过一个完整的实验，从部署一个 Pod 开始，一步一步地部署那些 Kubernetes 的组件，来剖析 Kubernetes 在网络层是如何实现及如何工作的。

这里使用虚拟机来完成实验。如果你要部署在物理机器上，或者部署在云服务商的环境下，则涉及的网络模型很可能稍微有所不同。不过，从网络角度来看，Kubernetes 的机制是类似且一致的。

好了，来看看我们的实验环境，如图 3.30 所示。

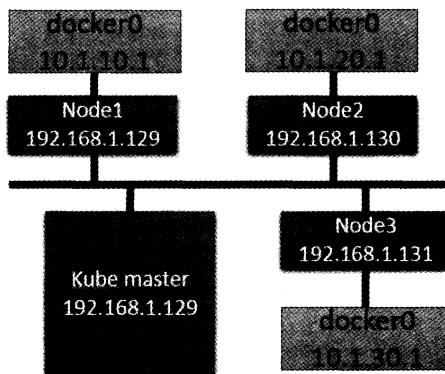


图 3.30 实验环境

Kubernetes 的网络模型要求每一个 Node 上的容器都可以相互访问。

默认的 Docker 的网络模型提供了一个 IP 地址段是 172.17.0.0/16 的 docker0 网桥。每一个容器都会在这个子网内获得 IP 地址，并且将 docker0 网桥的 IP 地址（172.17.42.1）作为其默认网关。需要注意的是 Docker 宿主机外面的网络不需要知道任何关于这个 172.17.0.0/16 的信息或者知道如何连接到它内部，因为 Docker 的宿主机针对容器发出的数据，在物理网卡地址后面都做了 IP 伪装 MASQUERADE（隐含 NAT）。也就是说，在网络上看到的任何容器数据流都来源于那台 Docker 节点的物理 IP 地址。这里所说的网络都是指连接这些主机的物理网络。

这个模型便于使用，但是并不完美，需要依赖端口映射的机制。

在 Kubernetes 的网络模型中，每台主机上的 docker0 网桥都是可以被路由到的。也就是说，在部署了一个 Pod 的时候，在同一个集群内，那台主机的外面可以直接访问到那个 Pod，并不

需要在那台物理主机上做端口映射。综上所述，你可以在网络层将 Kubernetes 的节点看作一个路由器。如果我们将实验环境改画成一个网络图，那么它看起来如图 3.31 所示。

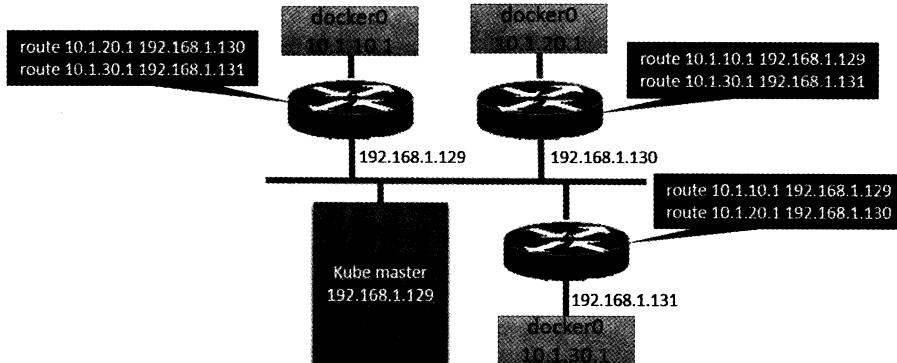


图 3.31 实验环境网络图

为了支持 Kubernetes 网络模型，我们采取了直接路由的方式来实现，在每个 Node 上配置相应的静态路由项，例如在 192.168.1.129 这个 Node 上我们配置了两个路由项：

```
# route add -net 10.1.20.0 netmask 255.255.255.0 gw 192.168.1.30
# route add -net 10.1.30.0 netmask 255.255.255.0 gw 192.168.1.31
```

这意味着，每一个新部署的容器都将使用这个 Node (docker0 的网桥 IP) 作为它的默认网关。而这些 Node 节点（类似路由器）都有其他 docker0 的路由信息，这样它们就能够相互连通了。

接下来通过一些实际的案例，来看看 Kubernetes 在不同的场景下其网络部分到底做了什么事情。

第1步：部署一个 RC/Pod

部署的 RC/Pod 描述文件如下 (frontend-controller.yaml)：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 1
  selector:
    name: frontend
  template:
    metadata:
```

```
labels:
  name: frontend
spec:
  containers:
  - name: php-redis
    image: kubeguide/guestbook-php-frontend
    env:
    - name: GET_HOSTS_FROM
      value: env
    ports:
    - containerPort: 80
      hostPort: 80
```

为了便于观察，我们假定在一个空的 Kubernetes 集群上运行，提前清理了所有 Replication Controller、Pod 和其他 Service：

```
# kubectl get rc
CONTROLLER  CONTAINER(S)  IMAGE(S)  SELECTOR  REPLICAS
#
# kubectl get services
NAME        LABELS           SELECTOR  IP(S)     PORT(S)
kubernetes  component=apiserver,provider=kubernetes <none>   20.1.0.1  443/TCP
#
# kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
```

让我们检查一下此时某个 Node 上的网络接口都有哪些。Node1 的状态是：

```
# ifconfig
docker0: flags=4099<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
  inet 10.1.10.1 netmask 255.255.255.0 broadcast 10.1.10.255
    inet6 fe80::5484:7aff:fe97:999 prefixlen 64 scopeid 0x20<link>
      ether 56:84:7a:fe:97:99 txqueuelen 0 (Ethernet)
      RX packets 373245 bytes 170175373 (162.2 MiB)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 353569 bytes 353948005 (337.5 MiB)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

eno16777736: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
  inet 192.168.1.129 netmask 255.255.255.0 broadcast 192.168.1.255
    inet6 fe80::20c:29ff:fe47:6e2c prefixlen 64 scopeid 0x20<link>
      ether 00:0c:29:47:6e:2c txqueuelen 1000 (Ethernet)
      RX packets 326552 bytes 286033393 (272.7 MiB)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 219520 bytes 31014871 (29.5 MiB)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
```

```

inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 0 (Local Loopback)
RX packets 24095 bytes 2133648 (2.0 MiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 24095 bytes 2133648 (2.0 MiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

可以看出，有一个 docker0 网桥和一个本地地址的网络端口。现在部署一下我们在前面准备的 RC/Pod 配置文件，看看发生了什么：

```

# kubectl create -f frontend-controller.yaml
replicationcontrollers/frontend
#
# kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE     NODE
frontend-4o11g 1/1     Running   0          11s    192.168.1.130

```

可以看到一些有趣的事情。Kubernetes 为这个 Pod 找了一个主机 192.168.1.130 (Node2) 来运行它。另外，这个 Pod 还获得了一个在 Node2 的 docker0 网桥上的 IP 地址。我们登录到 Node2 上看看发生了什么事情：

```

# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
37b193a4c633        kubeguide/example-guestbook-php-redis   "/bin/sh -c /run.sh"
32 seconds ago      Up 26 seconds       k8s_php-redis.6ad3289e_frontend-n9n1m_
development_813e2dd9-8149-11e5-823b-000c2921ba71_af6dd859
6d1b99cff4ae        google_containers/pause:latest    "/pause"           35 seconds ago
Up 28 seconds       0.0.0.0:80->80/tcp   k8s POD.855eeb3d_frontend-4t52y_development_
813e3870-8149-11e5-823b-000c2921ba71_2b66f05e

```

在 Node2 上现在运行了两个容器。在我们的 RC/Pod 定义文件中仅仅包含了一个，那么这第 2 个是从哪里来的呢？第 2 个看起来运行的是一个叫作 google_containers/pause:latest 的镜像，而且这个容器已经有端口映射到它上面了，为什么是这样呢？让我们深入容器内部去看一下具体原因。使用 Docker 的 “inspect” 命令来查看容器的详细信息，特别要关注容器的网络模型。

```

# docker inspect 6d1b99cff4ae | grep NetworkMode
    "NetworkMode": "bridge",
# docker inspect 37b193a4c633 | grep NetworkMode
    "NetworkMode": "container:6d1b99cff4ae537689ce87d7528f4ba9dbb40ae
711ecc0a5b3f7c39ff5e5e495",

```

有趣的结果是，在查看完每个容器的网络模型后，我们可以看到这样的配置：我们检查的第一个容器是运行了 “google_containers/pause:latest” 镜像的容器，它使用了 Docker 默认的网络模型 bridge；而我们检查的第 2 个容器，也就是在我们 RC/Pod 中定义运行的 php-redis 容器，使用了非默认的网络配置和映射容器的模型，指定了映射目标容器为 “google_containers/ pause:latest”。

我们一起来仔细思考一下这个过程，为什么 Kubernetes 要这么做呢？首先，一个 Pod 内的所有容器都需要共用同一个 IP 地址，这就意味着一定要使用网络的容器映射模式。然而，为什么不能只启动第 1 个 Pod 中的容器，而将第 2 个 Pod 内的容器关联到第 1 个容器呢？我们认为 Kubernetes 从两个方面来考虑这个问题：首先，如果 Pod 有超过两个容器的话，则连接这些容器可能不容易；其次，后面的容器还要依赖第 1 个被关联的容器，如果第 2 个容器关联到第 1 个容器，且第 1 个容器死掉的话，第 2 个也将死掉。启动一个基础容器，然后将 Pod 内的所有容器都连接到它上面会更容易一些。因为我们只需要为基础的这个 `Google_containers/pause` 容器执行端口映射规则，这也简化了端口映射的过程。所以我们的 Pod 的网络模型类似于图 3.32。

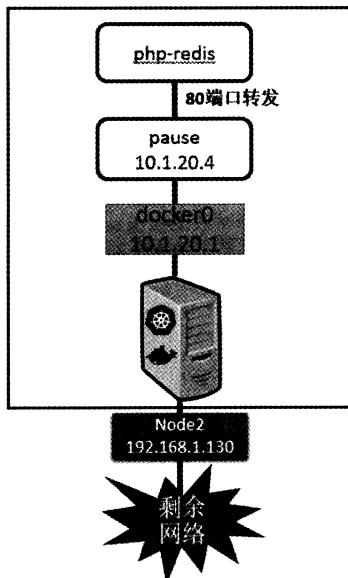


图 3.32 启动 Pod 后网络模型

在这种情况下，实际 Pod 的 IP 数据流的网络目标都是这个 `google_containers/pause` 容器。图 3.32 有点儿取巧地显示了是 `google_containers/pause` 容器将端口 80 的流量转发给了相关的容器。而 Pause 只是逻辑上的，并没有真的这么做。实际上另外的 Web 容器直接监听了这些端口，和 `google_containers/pause` 容器共享了同一个网络堆栈。这就是为什么 Pod 内部实际容器的端口映射都显示到 `google_containers/pause` 容器上了。我们可以通过 `docker port` 命令来检验一下：

```
# docker ps
CONTAINER ID        IMAGE
37b193a4c633      kubeguide/example-guestbook-php-redis
6d1b99cff4ae      google_containers/pause:latest
#
# docker port 6d1b99cff4ae
```

```
80/tcp -> 0.0.0.0:80
```

综上所述，`google_containers/pause` 容器实际上只是负责接管这个 Pod 的 Endpoint，它实际上并没有做更多的事情。那么 Node 呢，它需要将数据流传给 `google_containers/pause` 容器吗？我们来检查一下 Iptables 的规则，看看有什么发现：

```
# iptables-save
# Generated by iptables-save v1.4.21 on Thu Sep 24 17:15:01 2015
*nat
:PREROUTING ACCEPT [0:0]
:INPUT ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
:POSTROUTING ACCEPT [0:0]
:DOCKER - [0:0]
:KUBE-NODEPORT-CONTAINER - [0:0]
:KUBE-NODEPORT-HOST - [0:0]
:KUBE-PORTALS-CONTAINER - [0:0]
:KUBE-PORTALS-HOST - [0:0]
-A PREROUTING -m comment --comment "handle ClusterIPs; NOTE: this must be before
the NodePort rules" -j KUBE-PORTALS-CONTAINER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -m comment --comment "handle service
NodePorts; NOTE: this must be the last rule in the chain" -j KUBE-NODEPORT-CONTAINER
-A OUTPUT -m comment --comment "handle ClusterIPs; NOTE: this must be before the
NodePort rules" -j KUBE-PORTALS-HOST
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT -m addrtype --dst-type LOCAL -m comment --comment "handle service
NodePorts; NOTE: this must be the last rule in the chain"
-A POSTROUTING -s 10.1.20.0/24 ! -o docker0 -j MASQUERADE
-A KUBE-PORTALS-CONTAINER -d 20.1.0.1/32 -p tcp -m comment --comment
"default/kubernetes:" -m tcp --dport 443 -j REDIRECT --to-ports 60339
-A KUBE-PORTALS-HOST -d 20.1.0.1/32 -p tcp -m comment --comment
"default/kubernetes:" -m tcp --dport 443 -j DNAT --to-destination 192.168.1.131:60339
COMMIT
# Completed on Thu Sep 24 17:15:01 2015
# Generated by iptables-save v1.4.21 on Thu Sep 24 17:15:01 2015
*filter
:INPUT ACCEPT [1131:377745]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [1246:209888]
:DOCKER - [0:0]
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER -d 172.17.0.19/32 ! -i docker0 -o docker0 -p tcp -m tcp --dport 5000
-j ACCEPT
```

```
COMMIT  
# Completed on Thu Sep 24 17:15:01 2015
```

上面的这些规则并没有应用到我们刚刚定义的 Pod。当然，Kubernetes 会给每一个 Kubernetes 的节点提供一些默认的服务，上面的规则就是 Kubernetes 的默认服务需要的。关键是，我们没有看到任何 IP 伪装的规则，并且没有任何指向 Pod 10.1.20.4 的内部方向的端口映射。

第 2 步：发布一个服务

我们已经了解了 Kubernetes 如何处理最基本的元素 Pod 的连接问题，接下来看一下它是如何处理 Service 的。Service 允许我们在多个 Pod 之间抽象一些服务，而且，服务可以通过提供在同一个 Service 的多个 Pod 之间的负载均衡机制来支持水平扩展。我们再次将环境初始化，删除刚刚创建的 RC/Pod 来确保集群是空的：

```
# kubectl stop rc frontend  
replicationcontroller/frontend  
#  
# kubectl get rc  
CONTROLLER    CONTAINER(S)    IMAGE(S)    SELECTOR    REPLICAS  
#  
# kubectl get services  
NAME          LABELS                           SELECTOR    IP(S)      PORT(S)  
kubernetes    component=apiserver,provider=kubernetes  <none>     20.1.0.1  
443/TCP  
#  
# kubectl get pods  
NAME    READY    STATUS    RESTARTS   AGE
```

然后准备一个名称为 frontend 的 Service 配置文件：

```
apiVersion: v1  
kind: Service  
metadata:  
  name: frontend  
  labels:  
    name: frontend  
spec:  
  ports:  
    - port: 80  
  #    nodePort: 30001  
  selector:  
    name: frontend  
  #  type:  
  #  NodePort
```

然后在 Kubernetes 集群中定义这个服务：

```
# kubectl create -f frontend-service.yaml
services/frontend
# kubectl get services
NAME      LABELS            SELECTOR          IP(S)        PORT(S)
frontend   name=frontend    name=frontend     20.1.244.75  80/TCP
kubernetes component=apiserver,provider=kubernetes <none>       20.1.0.1
443/TCP
```

服务正确创建后，可以看到 Kubernetes 集群已经为这个服务分配了一个虚拟 IP 地址 20.1.244.75，这个 IP 地址是在 Kubernetes 的 Portal Network 中分配的。而这个 Portal Network 的地址范围则是我们在 Kubmaster 上启动 API 服务进程时，使用--service-cluster-ip-range=xx 命令行参数指定的：

```
# cat /etc/kubernetes/apiserver
...
# Address range to use for services
KUBE_SERVICE_ADDRESSES="--service-cluster-ip-range=20.1.0.0/16"
...
```

这个 IP 段可以是任何段，只要不和 docker0 或者物理网络的子网冲突就可以。选择任意其他网段的原因是这个网段将不会在物理网络和 docker0 网络上进行路由。这个 Portal Network 针对每一个 Node 都有局部的特殊性，实际上它存在的意义是让容器的流量都指向默认网关（也就是 docker0 网桥）。在继续实验前，先登录到 Node1 上看一下我们定义服务后发生了什么变化。首先检查一下 Iptables/Netfilter 的规则：

```
# iptables-save
...
-A KUBE-PORTRALES-CONTAINER -d 20.1.244.75/32 -p tcp -m comment --comment "default/
frontend:" -m tcp --dport 80 -j REDIRECT --to-ports 59528
-A KUBE-PORTRALES-HOST -d 20.1.244.75/32 -p tcp -m comment --comment "default/
kubernetes:" -m tcp --dport 80 -j DNAT --to-destination 192.168.1.131:59528
...
```

第 1 行是挂在 PREROUTING 链上的端口重定向规则，所有的进流量如果满足 20.1.244.75:80，则都会被重定向到端口 33761。第 2 行是挂在 OUTPUT 链上的目标地址 NAT，做了和上述第 1 行规则类似的工作，但针对的是当前主机生成的外出流量。所有主机生成的流量都需要使用这个 DNAT 规则来处理。简而言之，这两个规则使用了不同的方式做了类似的事情，就是将所有从节点生成的发送给 20.1.244.75:80 的流量重定向到本地的 33761 端口。

到此为止，目标为 Service IP 地址和端口的任何流量都将被重定向到本地的 33761 端口。这个端口连到哪里去了呢？这就到了 kube-proxy 发挥作用的地方了。这个 kube-proxy 服务给每一个新创建的服务关联了一个随机的端口号，并且监听那个特定的端口，为服务创建相关的负载均衡对象。在我们的实验中，随机生成的端口刚好是 33761。通过监控 Node1 上的 Kubernetes-Service 的日志，在创建服务时，我们可以看到下面的记录：

```
2612 proxier.go:413] Opened iptables from-containers portal for service "default/
frontend:" on TCP 20.1.244.75:80
2612 proxier.go:424] Opened iptables from-host portal for service "default/
frontend:" on TCP 20.1.244.75:80
```

现在我们知道，所有的流量都被导入 `kube-proxy`。现在我们需要它完成一些负载均衡的工作。创建 Replication Controller 并观察结果，下面是 Replication Controller 的配置文件：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  replicas: 3
  selector:
    name: frontend
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: php-redis
          image: kubeguide/example-guestbook-php-redis
          env:
            - name: GET_HOSTS_FROM
              value: env
          ports:
            - containerPort: 80
#           hostPort: 80
```

在集群发布上述配置文件后，等待并观察，确保所有 Pod 都运行起来了：

```
# kubectl create -f frontend-controller.yaml
replicationcontrollers/frontend
#
# kubectl get pods -o wide
NAME        READY   STATUS    RESTARTS   AGE       NODE
frontend-64t8q  1/1     Running   0          5s       192.168.1.130
frontend-dzqve  1/1     Running   0          5s       192.168.1.131
frontend-x5dwy  1/1     Running   0          5s       192.168.1.129
```

现在所有的 Pod 都运行起来了，Service 将会对匹配到标签为 “`name=frontend`” 的所有 Pod 进行负载分发。因为 Service 的选择匹配所有的这些 Pod，所以我们的负载均衡将会对这 3 个 Pod 进行分发。现在我们做实验的环境如图 3.33 所示。

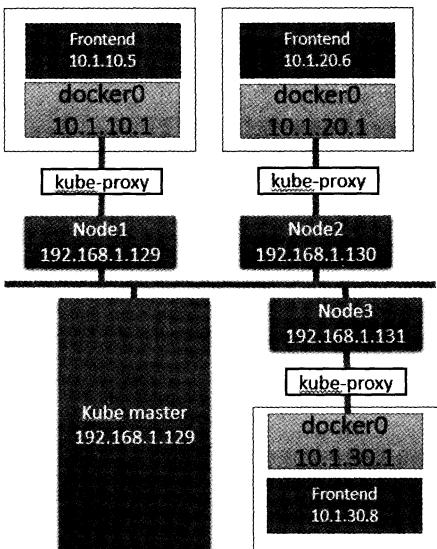


图 3.33 启动服务后的结构

Kubernetes 的 kube-proxy 看起来只是一个夹层,但实际上它只是在 Node 上运行的一个服务。上述重定向规则的结果就是针对目标地址为服务 IP 的流量,将 Kubernetes 的 kube-proxy 变成了一个中间的夹层。

为了查看具体的重定向动作,我们会使用 `tcpdump` 来进行网络抓包操作。首先,安装 `tcpdump`:

```
yum -y install tcpdump
```

安装完成后,登录 Node1,运行 `tcpdump` 命令:

```
tcpdump -nn -q -i eno1677736 port 80
```

需要捕获物理服务器以太网接口的数据包,Node1 机器上的以太网接口名字叫作 `eno1677736`。

再打开第 1 个窗口运行第 2 个 `tcpdump` 程序,不过我们需要一些额外的信息去运行它,即挂接在 `docker0` 桥上的虚拟网卡 Veth 的名字。我们看到只有一个 `frontend` 容器在 Node1 主机上运行,所以可以使用简单的“`ip addr`”命令来查看唯一的“Veth”网络接口:

```
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
```

```
2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP qlen 1000
    link/ether 00:0c:29:47:6e:2c brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.129/24 brd 192.168.1.255 scope global eno16777736
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe47:6e2c/64 scope link
        valid_lft forever preferred_lft forever
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
link/ether 56:84:7a:fe:97:99 brd ff:ff:ff:ff:ff:ff
inet 10.1.10.1/24 brd 10.1.10.255 scope global docker0
    valid_lft forever preferred_lft forever
inet6 fe80::5484:7aff:fefe:9799/64 scope link
    valid_lft forever preferred_lft forever
12: veth0558bfa: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
docker0 state UP
    link/ether 86:82:e5:c8:5a:9a brd ff:ff:ff:ff:ff:ff
    inet6 fe80::8482:e5ff:fec8:5a9a/64 scope link
        valid_lft forever preferred_lft forever
```

复制这个接口的名字，在第 2 个窗口中运行 tcpdump 命令。

```
tcpdump -nn -q -i veth0558bfa host 20.1.244.75
```

同时运行这两个命令，并且将窗口并排放置，以便同时看到两个窗口的输出：

```
# tcpdump -nn -q -i eno16777736 port 80
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eno16777736, link-type EN10MB (Ethernet), capture size 65535 bytes

# tcpdump -nn -q -i veth0558bfa host 20.1.244.75
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth0558bfa, link-type EN10MB (Ethernet), capture size 65535 bytes
```

好了，我们已经在同时捕获两个接口的网络包了。这时再启动第 3 个窗口，运行一个“`docker exec`”命令来连接到我们的“frontend”的容器内部（你可以先执行 `docker ps` 来获得这个容器的 ID）：

```
# docker ps
CONTAINER ID        IMAGE               ...
268ccdfb9524      kubeguide/example-guestbook-php-redis ...
6a519772b27e       google_containers/pause:latest ...
```

执行命令进入容器内部：

```
# docker exec -it 268ccdfb9524 bash
# docker exec -it 268ccdfb9524 bash
root@frontend-x5dwY:/#
```

一旦进入运行的容器内部，我们就可以通过 Pod 的 IP 地址来访问服务了。使用 curl 来尝试访问服务：

```
curl 20.1.244.75
```

在使用 curl 访问服务时，将在抓包的两个窗口内看到：

```
20:19:45.208948 IP 192.168.1.129.57452 > 10.1.30.8.8080: tcp 0
20:19:45.209005 IP 10.1.30.8.8080 > 192.168.1.129.57452: tcp 0
20:19:45.209013 IP 192.168.1.129.57452 > 10.1.30.8.8080: tcp 0
20:19:45.209066 IP 10.1.30.8.8080 > 192.168.1.129.57452: tcp 0

20:19:45.209227 IP 10.1.10.5.35225 > 20.1.244.75.80: tcp 0
20:19:45.209234 IP 20.1.244.75.80 > 10.1.10.5.35225: tcp 0
20:19:45.209280 IP 10.1.10.5.35225 > 20.1.244.75.80: tcp 0
20:19:45.209336 IP 20.1.244.75.80 > 10.1.10.5.35225: tcp 0
```

这些信息说明了什么问题呢？让我们在网络图上用实线标出第 1 个窗口中网络抓包信息的含义（物理网卡上的网络流量），并用虚线标出第 2 个窗口中网络抓包信息的含义（docker0 网桥上的网络流量），如图 3.34 所示。

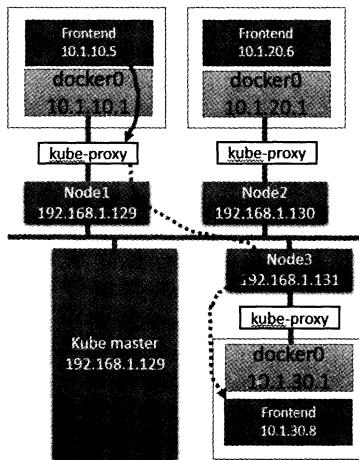


图 3.34 数据流动情况图 1

注意，图 3.34 中，虚线绕过了 Node3 的 kube-proxy，这么做是因为 Node3 上的 kube-proxy 没有参与这次网络交互。换句话说，Node1 的 kube-proxy 服务直接和负载均衡到的 Pod 进行网络交互。

在查看第 2 个捕获包的窗口时，我们能够站在容器的视角看这些流量。首先，容器尝试使用 20.1.244.75:80 打开 TCP 的 Socket 连接。同时，我们还可以看到从服务地址 20.1.244.75 返回的数据。从容器的视角来看，整个交互过程都是在服务之间进行的。但是在查看一个捕获包的窗口时（上面的窗口），我们可以看到物理机之间的数据交互，可以看到一个 TCP 连接从 Node1 的物理地址（192.168.1.129）发出，直接连接到运行 Pod 的主机 Node3（192.168.1.131）。总而言之，Kubernetes 的 kube-proxy 作为一个全功能的代理服务器管理了两个独立的 TCP 连接：一

个是从容器到 kube-proxy；另一个是从 kube-proxy 到负载均衡的目标 Pod。

如果我们清理一下捕获的记录，再次运行 curl，则还可以看到网络流量被负载均衡转发到另一个节点 Node2 上了。

```
20:19:45.208948 IP 192.168.1.129.57485 > 10.1.20.6.8080: tcp 0
20:19:45.209005 IP 10.1.20.6.8080 > 192.168.1.129.57485: tcp 0
20:19:45.209013 IP 192.168.1.129.57485 > 10.1.20.6.8080: tcp 0
20:19:45.209066 IP 10.1.20.6.8080 > 192.168.1.129.57485: tcp 0

20:19:45.209227 IP 10.1.10.5.38026 > 20.1.244.75.80: tcp 0
20:19:45.209234 IP 20.1.244.75.80 > 10.1.10.5.38026: tcp 0
20:19:45.209280 IP 10.1.10.5.38026 > 20.1.244.75.80: tcp 0
20:19:45.209336 IP 20.1.244.75.80 > 10.1.10.5.38026: tcp 0
```

这一次，Kubernetes 的 Proxy 将选择运行在 Node2 (10.1.20.1) 上面的 Pod 作为负载均衡的目的。网络流动图如图 3.35 所示。

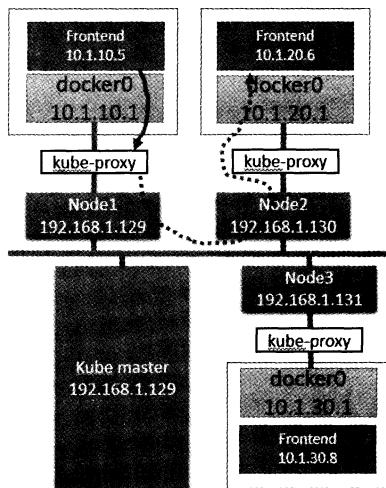


图 3.35 数据流动情况图 2

到这里，你肯定已经知道另外一个可能的负载均衡的路由结果了吧。

第4章

Kubernetes 开发指南

本章将引入 REST 的概念, 详细说明 Kubernetes API, 并举例说明如何基于 Jersey 和 Fabric8 框架访问 Kubernetes API, 深入分析基于这两个框架访问 Kubernetes API 的优缺点。下面从 REST 开始说起。

4.1 REST 简述

REST(Representational State Transfer)是由 Roy Thomas Fielding 博士在他的论文 *Architectural Styles and the Design of Network-based Software Architectures* 中提出的一个术语。REST 本身只是为分布式超媒体系统设计的一种架构风格, 而不是标准。

基于 Web 的架构实际上就是各种规范的集合, 这些规范共同组成了 Web 架构, 比如 HTTP、客户端服务器模式都是规范。每当我们原有规范的基础上增加新的规范时, 就会形成新的架构。而 REST 正是这样一种架构, 它结合了一系列规范, 形成了一种新的基于 Web 的架构风格。

传统的 Web 应用大多是 B/S 架构, 涉及如下规范。

(1) 客户-服务器: 这种规范的提出, 改善了用户接口跨多个平台的可移植性, 并且通过简化服务器组件, 改善了系统的可伸缩性。最为关键的是通过分离用户接口和数据存储, 使得不同的用户终端共享相同的数据成为可能。

(2) 无状态性: 无状态性是在客户-服务器约束的基础上添加的又一层规范, 它要求通信必须在本质上是无状态的, 即从客户端到服务器的每个 request 都必须包含理解该 request 所必需

的所有信息。这个规范改善了系统的可见性（无状态性使得客户端和服务器端不必保存对方的详细信息，服务器只需要处理当前的 `request`，而不必了解所有 `request` 的历史）、可靠性（无状态性减少了服务器从局部错误中恢复的任务量）、可伸缩性（无状态性使得服务器端可以很容易地释放资源，因为服务器端不必在多个 `request` 中保存状态）。同时，这种规范的缺点也是显而易见的，由于不能将状态数据保存在服务器上，因此增加了在一系列 `request` 中发送重复数据的开销，严重降低了效率。

（3）缓存：为了改善无状态性带来的网络的低效性，我们添加了缓存约束。缓存约束允许隐式或显式地标记一个 `response` 中的数据，赋予了客户端缓存 `response` 数据的功能，这样就可以为以后的 `request` 共用缓存的数据，部分或全部地消除一部分交互，提高了网络效率。但是由于客户端缓存了信息，所以增加了客户端与服务器数据不一致的可能性，从而降低了可靠性。

B/S 架构的优点是部署非常方便，在用户体验方面却不很理想。为了改善这种情况，我们引入了 REST。REST 在原有架构上增加了三个新规范：统一接口、分层系统和按需代码。

（1）统一接口：REST 架构风格的核心特征就是强调组件之间有一个统一的接口，表现为在 REST 世界里，网络上的所有事物都被抽象为资源，REST 通过通用的连接器接口对资源进行操作。这样设计的好处是保证系统提供的服务都是解耦的，极大地简化了系统，从而改善了系统的交互性和可重用性。

（2）分层系统：分层系统规则的加入提高了各种层次之间的独立性，为整个系统的复杂性设置了边界，通过封装遗留的服务，使新的服务器免受遗留客户端的影响，也提高了系统的可伸缩性。

（3）按需代码：REST 允许对客户端功能进行扩展。比如，通过下载并执行 `applet` 或脚本形式的代码来扩展客户端的功能。但这在改善系统可扩展性的同时降低了可见性，所以它只是 REST 的一个可选约束。

REST 架构是针对 Web 应用而设计的，其目的是为了降低开发的复杂性，提高系统的可伸缩性。REST 提出了如下设计准则。

- （1）网络上的所有事物都被抽象为资源（Resource）。
- （2）每个资源对应一个唯一的资源标识符（Resource Identifier）。
- （3）通过通用的连接器接口（Generic Connector Interface）对资源进行操作。
- （4）对资源的各种操作不会改变资源标识符。
- （5）所有的操作都是无状态的（Stateless）。

REST 中的资源所指的不是数据，而是数据和表现形式的组合，比如“最新访问的 10 位会员”和“最活跃的 10 位会员”在数据上可能有重叠或者完全相同，而由于它们的表现形式不同，所以被归为不同的资源，这也就是为什么 REST 的全名是 Representational State Transfer。资源标识符就是 URI (Uniform Resource Identifier)，不管是图片、Word 还是视频文件，甚至只是一种虚拟的服务，也不管是 xml、txt 还是其他文件格式，全部通过 URI 对资源进行唯一标识。

REST 是基于 HTTP 的，任何对资源的操作行为都通过 HTTP 来实现。以往的 Web 开发大多数用的是 HTTP 中的 GET 和 POST 方法，很少使用其他方法，这实际上是因为对 HTTP 的片面理解造成的。HTTP 不仅仅是一个简单的运载数据的协议，而且是一个具有丰富内涵的网络软件的协议，它不仅能对互联网资源进行唯一定位，还能告诉我们如何对该资源进行操作。HTTP 把对一个资源的操作限制在 4 种方法内：GET、POST、PUT 和 DELETE，这正是对资源 CRUD 操作的实现。由于资源和 URI 是一一对应的，在执行这些操作时 URI 没有变化，和以往的 Web 开发有很大的区别，所以极大地简化了 Web 开发，也使得 URI 可以被设计成更为直观地反映资源的结构。这种 URI 的设计被称作 RESTful 的 URI，为开发人员引入了一种新的思维方式：通过 URL 来设计系统结构。当然了，这种设计方式对于一些特定情况也是不适用的，也就是说不是所有 URI 都适用于 RESTful。

REST 之所以可以提高系统的可伸缩性，就是因为它要求所有操作都是无状态的。由于没有了上下文 (Context) 的约束，做分布式和集群时就更为简单，也可以让系统更为有效地利用缓冲池 (Pool)，并且由于服务器端不需要记录客户端的一系列访问，也就减少了服务器端的性能损耗。

Kubernetes API 也符合 RESTful 规范，下面对其进行介绍。

4.2 Kubernetes API 详解

4.2.1 Kubernetes API 概述

Kubernetes API 是集群系统中的重要组成部分，Kubernetes 中各种资源（对象）的数据通过该 API 接口被提交到后端的持久化存储 (etcd) 中，Kubernetes 集群中的各部件之间通过该 API 接口实现解耦合，同时 Kubernetes 集群中一个重要且便捷的管理工具 kubectl 也是通过访问该 API 接口实现其强大的管理功能的。Kubernetes API 中的资源对象都拥有通用的元数据，资源对象也可能存在嵌套现象，比如在一个 Pod 里面嵌套多个 Container。创建一个 API 对象是指通过 API 调用创建一条有意义的记录，该记录一旦被创建，Kubernetes 将确保对应的资源对象会被

自动创建并托管维护。

在 Kubernetes 系统中，大多数情况下，API 定义和实现都符合标准的 HTTP REST 格式，比如通过标准的 HTTP 动词（POST、PUT、GET、DELETE）来完成对相关资源对象的查询、创建、修改、删除等操作。但同时 Kubernetes 也为某些非标准的 REST 行为实现了附加的 API 接口，例如 Watch 某个资源的变化、进入容器执行某个操作等。另外，某些 API 接口可能违背严格的 REST 模式，因为接口不是返回单一的 JSON 对象，而是返回其他类型的数据，比如 JSON 对象流（Stream）或非结构化的文本日志数据等。

Kubernetes 开发人员认为，任何成功的系统都会经历一个不断成长和不断适应各种变更的过程。因此，他们期望 Kubernetes API 是不断变更和增长的。同时，他们在设计和开发时，有意识地兼容了已存在的客户需求。通常，新的 API 资源（Resource）和新的资源域不希望被频繁地加入系统。资源或域的删除需要一个严格的审核流程。

为了方便查阅 API 接口的详细定义，Kubernetes 使用了 swagger-ui 提供 API 在线查询功能，其官网为 http://kubernetes.io/third_party/swagger-ui/，Kubernetes 开发团队会定期更新、生成 UI 及文档。Swagger UI 是一款 REST API 文档在线自动生成和功能测试软件，关于 Swagger 的内容请访问官网 <http://swagger.io>。

运行在 Master 节点上的 API Server 进程同时提供了 swagger-ui 的访问地址：`http://<master-ip>:<master-port>/swagger-ui/`。假设我们的 API Server 安装在 192.168.1.128 服务器上，绑定了 8080 端口，则可以通过访问 <http://192.168.1.128:8080/swagger-ui/> 来查看 API 信息，如图 4.1 所示。

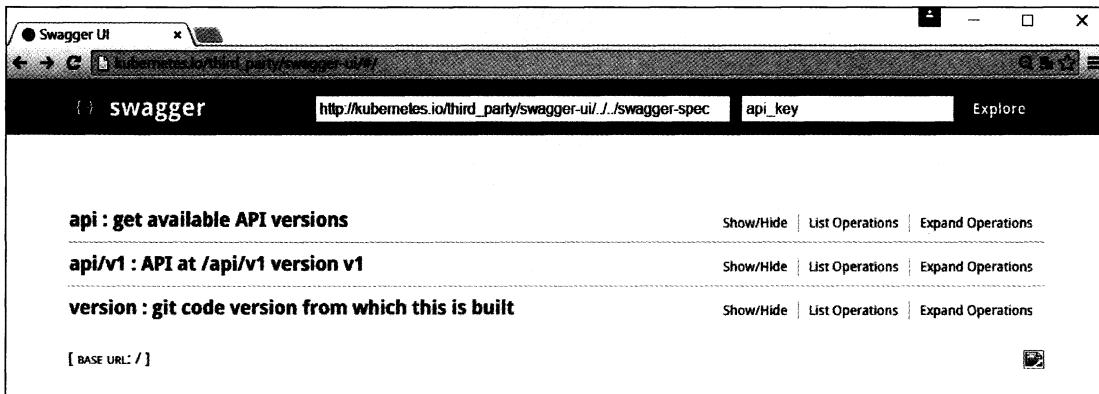


图 4.1 swagger-ui

单击 api/v1 可以查看所有 API 的列表，如图 4.2 所示。

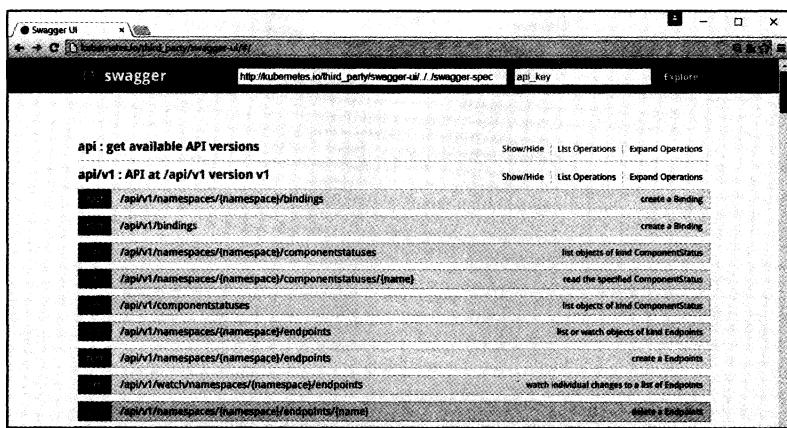


图 4.2 查看 API 列表

以 create a Pod 为例，找到 Rest API 的访问路径为：/api/v1/namespaces/{namespace}/pods，如图 4.3 所示。



图 4.3 Create a Pod API

单击链接展开，即可查看详细的 API 接口说明，如图 4.4 所示。

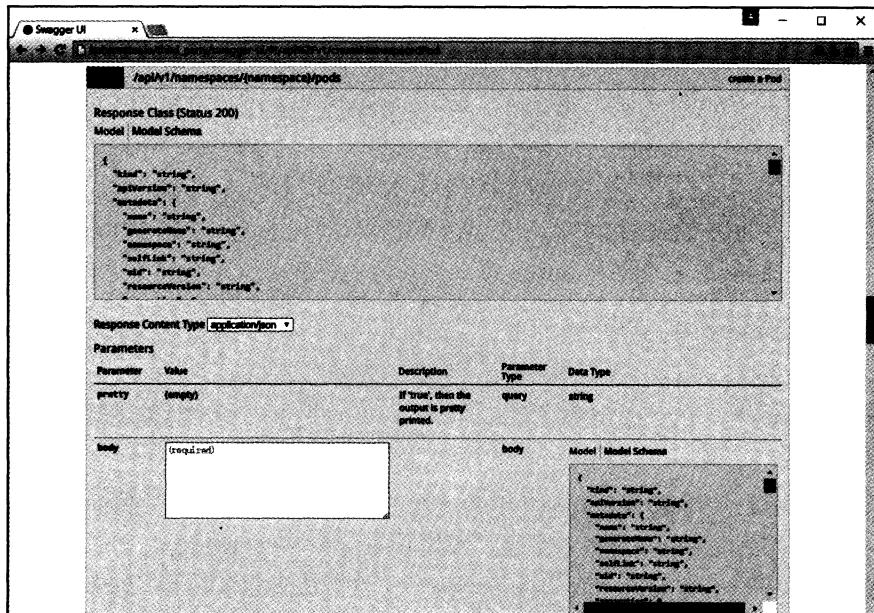


图 4.4 Create a Pod API 详细说明

单击 Model 链接，则可以查看文本格式显示的 API 接口描述，如图 4.5 所示。

```

POST /api/v1/namespaces/{namespace}/pods
      create a Pod

Response Class (Status 200)
Model Model Schema

v1.Pod {
  kind (string, optional): kind of object, in CamelCase; cannot be updated; see http://releases.k8s.io/HEAD/docs/api-conventions.md#types-kinds,
  apiVersion (string, optional): version of the schema the object should have; see http://releases.k8s.io/HEAD/docs/api-conventions.md#resources,
  metadata (v1.ObjectMeta, optional),
  spec (v1.PodSpec, optional),
  status (v1.PodStatus, optional)
}

v1.ObjectMeta {
  name (string, optional): string that identifies an object. Must be unique within a namespace; cannot be updated; see http://releases.k8s.io/HEAD/docs/identifiers.md#names.
  generateName (string, optional): an optional prefix to use to generate a unique name; has the same validation rules as name; optional, and is applied only if name is not specified; see http://releases.k8s.io/HEAD/docs/api-conventions.md#idempotency.
  namespace (string, optional): namespace of the object; must be a DNS_LABEL; cannot be updated; see http://releases.k8s.io/HEAD/docs/namespaces.md.
  selfLink (string, optional): URL for the object; populated by the system, read-only,
  uid (string, optional): unique UUID across space and time; populated by the system, read-only; see http://releases.k8s.io/HEAD/docs/identifiers.md#uids.
  resourceVersion (string, optional): string that identifies the internal version of this object that can be used by clients to determine when objects have changed; populated by the system, read-only; value must be treated as opaque by clients and passed unmodified back to the server; see http://releases.k8s.io/HEAD/docs/api-conventions.md#concurrency-control-and-consistency.
  generation (integer, optional): a sequence number representing a specific generation of the desired state; populated by the system; read-only,
  creationTimestamp (string, optional): RFC 3339 date and time at which the object was created; populated by the system, read-only; null for lists; see http://releases.k8s.io/HEAD/docs/api-conventions.md#metadata.
  deletionTimestamp (string, optional): RFC 3339 date and time at which the object will be deleted; populated by the system when a graceful deletion is requested, read-only; if not set, graceful deletion of the object has not been requested; see http://releases.k8s.io/HEAD/docs/api-conventions.md#metadata.
  labels (undefined, optional),
  annotations (undefined, optional)
}

v1.PodSpec {
  volumes (Array[v1.Volume], optional): list of volumes that can be mounted by containers belonging to the pod; see http://releases.k8s.io/HEAD/docs/volumes.md.
  containers (Array[v1.Container]): list of containers belonging to the pod; cannot be updated; containers cannot currently be added or removed; there must be at least one container in a Pod; see http://releases.k8s.io/HEAD/docs/containers.md.
  restartPolicy (string, optional): restart policy for all containers within the pod; one of Always, OnFailure, Never; defaults to Always; see http://releases.k8s.io/HEAD/docs/pod-states.md#restartpolicy.
  terminationGracePeriodSeconds (integer, optional): optional duration in seconds the pod needs to terminate gracefully; may be decreased in delete request; value must be non-negative integer; the value zero indicates delete immediately; if this value is not set, the default grace period will be used instead; the grace period is the duration in seconds after the processes running in the pod are sent a termination signal and the time when the processes are forcibly halted with
}

```

图 4.5 Create a Pod API 文本格式详细说明

我们看到，在 Kubernetes API 中，一个 API 的顶层（Top Level）元素由 kind、apiVersion、metadata、spec 和 status 等几个部分组成，接下来，我们分别对这几个部分进行说明。

kind 表明对象有以下三大类别。

(1) 对象 (objects)：代表在系统中的一个永久资源（实体），例如 Pod、RC、Service、Namespace 及 Node 等。通过操作这些资源的属性，客户端可以对该对象进行创建、修改、删除和获取操作。

(2) 列表 (list)：一个或多个资源类别的集合。列表有一个通用元数据的有限集合。所有列表 (lists) 通过“items”域获得对象数组，例如 PodLists、ServiceLists、NodeLists。大部分定义在系统中的对象都有一个返回所有资源 (resource) 集合的端点，以及零到多个返回所有资源集合的子集的端点。某些对象有可能是单例对象 (singletons)，例如当前用户、系统默认用户等，

这些对象没有列表。

(3) 简单类别 (simple): 该类别包含作用在对象上的特殊行为和非持久实体。该类别限制了使用范围，它有一个通用元数据的有限集合，例如 Binding、Status。

apiVersion 表明 API 的版本号，当前版本默认只支持 v1。

Metadata 是资源对象的元数据定义，是集合类的元素类型，包含一组由不同名称定义的属性。在 Kubernetes 中每个资源对象都必须包含以下 3 种 Metadata。

(1) namespace: 对象所属的命名空间，如果不指定，系统则会将对象置于名为“default”的系统命名空间中。

(2) name: 对象的名字，在一个命名空间中名字应具备唯一性。

(3) uid: 系统为每个对象生成的唯一 ID，符合 RFC 4122 规范的定义。

此外，每种对象还应该包含以下几个重要元数据。

(1) labels: 用户可定义的“标签”，键和值都为字符串的 map，是对对象进行组织和分类的一种手段，通常用于标签选择器 (Label Selector)，用来匹配目标对象。

(2) annotations: 用户可定义的“注解”，键和值都为字符串的 map，被 Kubernetes 内部进程或者某些外部工具使用，用于存储和获取关于该对象的特定元数据。

(3) resourceVersion: 用于识别该资源内部版本号的字符串，在用于 Watch 操作时，可以避免在 GET 操作和下一次 Watch 操作之间造成的信息不一致，客户端可以用它来判断资源是否改变。该值应该被客户端看作不透明，且不做任何修改就返回给服务端。客户端不应该假定版本信息具有跨命名空间、跨不同资源类别、跨不同服务器的含义。

(4) creationTimestamp: 系统记录创建对象时的时间戳，符合 RFC 3339 规范。

(5) deletionTimestamp: 系统记录删除对象时的时间戳，符合 RFC 3339 规范。

(6) selfLink: 通过 API 访问资源自身的 URL，例如一个 Pod 的 link 可能是/api/v1/namespaces/default/pods/frontend-08bg4。

spec 是集合类的元素类型，用户对需要管理的对象进行详细描述的主体部分都在 spec 里给出，它会被 Kubernetes 持久化到 etcd 中保存，系统通过 spec 的描述来创建或更新对象，以达到用户期望的对象运行状态。spec 的内容既包括用户提供的配置设置、默认值、属性的初始化值，也包括在对象创建过程中由其他相关组件（例如 schedulers、auto-scalers）创建或修改的对象属性，比如 Pod 的 Service IP 地址。如果 spec 被删除，那么该对象将会从系统中被删除。

Status 用于记录对象在系统中的当前状态信息，它也是集合类元素类型，status 在一个自动处理的进程中被持久化，可以在流转的过程中生成。如果观察到一个资源丢失了它的状态

(Status)，则该丢失的状态可能被重新构造。以 Pod 为例，Pod 的 status 信息主要包括 conditions、containerStatuses、hostIP、phase、podIP、startTime 等。其中比较重要的两个状态属性如下。

(1) phase：描述对象所处的生命周期阶段，phase 的典型值是“Pending（创建中）”“Running”“Active（正在运行中）”或“Terminated（已终结）”，这几种状态对于不同的对象可能有轻微的差别，此外，关于当前 phase 附加的详细说明可能包含在其他域中。

(2) condition：表示条件，由条件类型和状态值组成，目前仅有一种条件类型 Ready，对应的状态值可以为 True、False 或 Unknown。一个对象可以具备多种 condition，而 condition 的状态值也可能不断发生变化，condition 可能附带一些信息，例如最后的探测时间或最后的转变时间。

4.2.2 API 版本

为了在兼容旧版本的同时不断升级新的 API，Kubernetes 提供了多版本 API 的支持能力，每个版本的 API 通过一个版本号路径前缀进行区分，例如/api/v1beta3。通常情况下，新旧几个不同的 API 版本都能涵盖所有的 Kubernetes 资源对象，在不同的版本之间这些 API 接口存在一些细微差别。Kubernetes 开发团队基于 API 级别选择版本而不是基于资源和域级别，是为了确保 API 能够描述一个清晰的连续的系统资源和行为的视图，能够控制访问的整个过程和控制实验性 API 的访问。

API 及版本发布建议描述了版本升级的当前思路。版本 v1beta1、v1beta2 和 v1beta3 为不建议使用 (Deprecated) 的版本，请尽快转到 v1 版本。在 2015 年 6 月 4 日，Kubernetes v1 版本 API 正式发布。版本 v1beta1 和 v1beta2 API 在 2015 年 6 月 1 日被删除，版本 v1beta3 API 在 2015 年 7 月 6 日被删除。

4.2.3 API 详细说明

API 资源使用 REST 模式，具体说明如下。

(1) GET /<资源名的复数格式>：获得某一类型的资源列表，例如 GET /pods 返回一个 Pod 资源列表。

(2) POST /<资源名的复数格式>：创建一个资源，该资源来自用户提供的 JSON 对象。

(3) GET /<资源名复数格式>/<名字>：通过给出的名称 (Name) 获得单个资源，例如 GET /pods/first 返回一个名称为“first”的 Pod。

(4) DELETE /<资源名复数格式>/<名字>：通过给出的名字删除单个资源，在删除选项 (DeleteOptions) 中可以指定优雅删除 (Grace Deletion) 的时间 (GracePeriodSeconds)，该可选项表明了从服务端接收到删除请求到资源被删除的时间间隔 (单位为秒)。不同的类别 (Kind)

可能为优雅删除时间（Grace Period）申明默认值。用户提交的优雅删除时间将覆盖该默认值，包括值为 0 的优雅删除时间。

(5) PUT /<资源名复数格式>/<名字>：通过给出的资源名和客户端提供的 JSON 对象来更新或创建资源。

(6) PATCH /<资源名复数格式>/<名字>：选择修改资源详细指定的域。

对于 PATCH 操作，目前 Kubernetes API 通过相应的 HTTP 首部“Content-Type”对其进行识别。

目前支持以下三种类型的 PATCH 操作。

(1) JSON Patch, Content-Type: application/json-patch+json。在 RFC6902 的定义中，JSON Patch 是执行在资源对象上的一系列操作，例如 `{"op": "add", "path": "/a/b/c", "value": ["foo", "bar"]}`。详情请查看 RFC6902 说明，网址为 <HTTP://tools.ietf.org/html/rfc6902>。

(2) Merge Patch, Content-Type: application/merge-json-patch+json。在 RFC7386 的定义中，Merge Patch 必须包含对一个资源对象的部分描述，这个资源对象的部分描述就是一个 JSON 对象。该 JSON 对象被提交到服务端，并和服务端的当前对象合并，从而创建一个新的对象。详情请查看 RFC7386 说明，网址为 <HTTP://tools.ietf.org/html/rfc7386>。

(3) Strategic Merge Patch, Content-Type: application/strategic-merge-patch+json。Strategic Merge Patch 是一个定制化的 Merge Patch 实现。接下来将详细讲解 Strategic Merge Patch。

在标准的 JSON Merge Patch 中，JSON 对象总是被合并（merge）的，但是资源对象中的列表域总是被替换的。通常这不是用户所希望的。例如，我们通过下列定义创建一个 Pod 资源对象：

```
spec:
  containers:
    - name: nginx
      image: nginx-1.0
```

接着我们希望添加一个容器到这个 Pod 中，代码和上传的 JSON 对象如下所示：

```
PATCH /api/v1/namespaces/default/pods/pod-name
spec:
  containers:
    - name: log-tailer
      image: log-tailer-1.0
```

如果我们使用标准的 Merge Patch，则其中的整个容器列表将被单个的“log-tailer”容器所替换。然而我们的目的是两个容器列表能够合并。

为了解决这个问题，Strategic Merge Patch 通过添加元数据到 API 对象中，并通过这些新元数据来决定哪个列表被合并，哪个列表不被合并。当前这些元数据作为结构标签，对于 API 对象自身来说是合法的。对于客户端来说，这些元数据作为 Swagger annotations 也是合法的。在

上述例子中，向“containers”中添加“patchStrategy”域，且它的值为“merge”，通过添加“patchMergeKey”，它的值为“name”。也就是说，“containers”中的列表将会被合并而不是替换，合并的依据为“name”域的值。

此外，Kubernetes API 添加了资源变动的“观察者”模式的 API 接口。

- ◎ GET /watch/<资源名复数格式>：随时间变化，不断接收一连串的 JSON 对象，这些 JSON 对象记录了给定资源类别内所有资源对象的变化情况。
- ◎ GET /watch/<资源名复数格式>/<name>：随时间变化，不断接收一连串的 JSON 对象，这些 JSON 对象记录了某个给定资源对象的变化情况。

上述接口改变了返回数据的基本类别，watch 动词返回的是一连串的 JSON 对象，而不是单个的 JSON 对象。并不是所有的对象类别都支持“观察者”模式的 API 接口，在后续的章节中将会说明哪些资源对象支持这种接口。

另外，Kubernetes 还增加了 HTTP Redirect 与 HTTP Proxy 这两种特殊的 API 接口，前者实现资源重定向访问，后者则实现 HTTP 请求的代理。

4.2.4 API 响应说明

API Server 响应用户请求时附带一个状态码，该状态码符合 HTTP 规范。表 4.1 列出了 API Server 可能返回的状态码。

表 4.1 API Server 可能返回的状态码

状态码	编 码	描 述
200	OK	表明请求完全成功
201	Created	表明创建类的请求完全成功
204	NoContent	表明请求完全成功，同时 HTTP 响应不包含响应体。 在响应 OPTIONS 方法的 HTTP 请求时返回
307	TemporaryRedirect	表明请求资源的地址被改变，建议客户端使用 Location 首部给出的临时 URL 来定位资源
400	BadRequest	表明请求是非法的，建议客户不要重试，修改该请求
401	Unauthorized	表明请求能够到达服务端，且服务端能够理解用户请求，但是拒绝做更多的事情，因为客户端必须提供认证信息。如果客户端提供了认证信息，则返回该状态码，表明服务端指出所提供的认证信息不合适或非法
403	Forbidden	表明请求能够到达服务端，且服务端能够理解用户请求，但是拒绝做更多的事情，因为该请求被设置成拒绝访问。建议客户不要重试，修改该请求
404	NotFound	表明所请求的资源不存在。建议客户不要重试，修改该请求
405	MethodNotAllowed	表明请求中带有该资源不支持的方法。建议客户不要重试，修改该请求

续表

状态码	编 码	描 述
409	Conflict	表明客户端尝试创建的资源已经存在，或者由于冲突请求的更新操作不能被完成
422	UnprocessableEntity	表明由于所提供的作为请求部分的数据非法，创建或修改操作不能被完成
429	TooManyRequests	表明超出了客户端访问频率的限制或者服务端接收到多于它能处理的请求。建议客户端读取相应的 Retry-After 首部，然后等待该首部指出的时间后再重试
500	InternalServerError	表明服务端能被请求访问到，但是不能理解用户的请求；或者服务端内产生非预期中的一个错误，而且该错误无法被认知；或者服务端不能在一个合理的时间内完成处理（这可能由于服务器临时负载过重造成或者由于和其他服务器通信时的一个临时通信故障造成）
503	ServiceUnavailable	表明被请求的服务无效。建议客户不要重试修改该请求
504	ServerTimeout	表明请求在给定的时间内无法完成。客户端仅在为请求指定超时（Timeout）参数时会得到该响应

在调用 API 接口发生错误时，Kubernetes 将会返回一个状态类别（Status Kind）。下面是两种常见的错误场景：

- (1) 当一个操作不成功时（例如，当服务端返回一个非 2xx HTTP 状态码时）；
- (2) 当一个 HTTP DELETE 方法调用失败时。

状态对象被编码成 JSON 格式，同时该 JSON 对象被作为请求的响应体。该状态对象包含人和机器使用的域，这些域中包含来自 API 的关于失败原因的详细信息。状态对象中的信息补充了对 HTTP 状态码的说明。例如：

```
$ curl -v -k -H "Authorization: Bearer WhCDvq4VPpYhrcfmF6ei7V9qlbqTubUc"
HTTPs://10.240.122.184:443/api/v1/namespaces/default/pods/grafana
> GET /api/v1/namespaces/default/pods/grafana HTTP/1.1
> User-Agent: curl/7.26.0
> Host: 10.240.122.184
> Accept: /*
> Authorization: Bearer WhCDvq4VPpYhrcfmF6ei7V9qlbqTubUc
>

< HTTP/1.1 404 Not Found
< Content-Type: application/json
< Date: Wed, 20 May 2015 18:10:42 GMT
< Content-Length: 232
<
{
    "kind": "Status",
    "apiVersion": "v1",
    "metadata": {},
    "status": "Failure",
    "message": "pods \"grafana\" not found",
    "reason": "NotFound"
}
```

```
"reason": "NotFound",
"details": {
    "name": "grafana",
    "kind": "pods"
},
"code": 404
}
```

- ◎ “status” 域包含两个可能的值：Success 和 Failure。
- ◎ “message” 域包含对错误的可读描述。
- ◎ “reason” 域包含说明该操作失败原因的可读描述。如果该域的值为空，则表示该域内没有任何说明信息。“reason” 域澄清 HTTP 状态码，但没有覆盖该状态码。
- ◎ “details” 可能包含和 “reason” 域相关的扩展数据。每个 “reason” 域可以定义它的扩展的 “details” 域。该域是可选的，返回数据的格式是不确定的，不同的 reason 类型返回的 “details” 域的内容不一样。

4.3 使用 Java 程序访问 Kubernetes API

本节介绍如何使用 Java 程序访问 Kubernetes API。在 Kubernetes 的官网上列出了多个访问 Kubernetes API 的开源项目，其中有两个是用 Java 语言开发工具的开源项目，一个是 OSGI，另一个是 Fabric8。在本节所列的两个 Java 开发例子中，一个是基于 Jersey 的，另一个是基于 Fabric8 的。

4.3.1 Jersey

Jersey 是一个 RESTful 请求服务 JAVA 框架。与 Struts 类似，它可以和 Hibernate、Spring 框架整合。通过它不仅方便开发 RESTful Web Service，而且可以将它作为客户端方便地访问 RESTful Web Service 服务端。

如果没有一个好的工具包，则开发一个能够用不同的媒介（Media）类型无缝地暴露你的数据，以及很好地抽象客户、服务端通信的底层通信的 RESTful Web Services，会很不容易。为了能够简化用 Java 开发 RESTful Web Service 及其客户端的流程，业界设计了 JAX-RS API。Jersey RESTful Web Services 框架是一个开源的高质量的框架，它为用 JAVA 语言开发 RESTful Web Service 及其客户端而生，支持 JAX-RS APIs。Jersey 不仅支持 JAX-RS APIs，而且在此基础上扩展了 API 接口，这些扩展更加方便和简化了 RESTful Web Services 及其客户端的开发。

由于 Kuberetes API Server 是 RESTful Web Service，因此此处选用 Jersey 框架开发 RESTful

Web Service 客户端，用来访问 Kubernetes API。在本例中选用的 Jersey 框架的版本为 1.19，所涉及的 Jar 包如图 4.6 所示。

 commons-codec-1.2.jar	2015/9/13 11:10	Executable Jar File	30 KB
 commons-httpclient-3.1.jar	2015/9/13 11:09	Executable Jar File	298 KB
 commons-logging-1.0.4.jar	2015/9/13 11:10	Executable Jar File	38 KB
 jackson-core-asl-1.9.2.jar	2015/2/11 5:41	Executable Jar File	223 KB
 jackson-jaxrs-1.9.2.jar	2015/2/11 5:41	Executable Jar File	18 KB
 jackson-mapper-asl-1.9.2.jar	2015/2/11 5:41	Executable Jar File	748 KB
 jackson-xc-1.9.2.jar	2015/2/11 5:41	Executable Jar File	27 KB
 jersey-apache-client-1.19.jar	2015/2/11 5:41	Executable Jar File	22 KB
 jersey-atom-abdera-1.19.jar	2015/2/11 5:41	Executable Jar File	20 KB
 jersey-client-1.19.jar	2015/2/11 5:41	Executable Jar File	131 KB
 jersey-core-1.19.jar	2015/2/11 5:41	Executable Jar File	427 KB
 jersey-guice-1.19.jar	2015/2/11 5:41	Executable Jar File	16 KB
 jersey-json-1.19.jar	2015/2/11 5:41	Executable Jar File	162 KB
 jersey-multipart-1.19.jar	2015/2/11 5:41	Executable Jar File	53 KB
 jersey-server-1.19.jar	2015/2/11 5:41	Executable Jar File	687 KB
 jersey-servlet-1.19.jar	2015/2/11 5:41	Executable Jar File	126 KB
 jersey-simple-server-1.19.jar	2015/2/11 5:41	Executable Jar File	12 KB
 jersey-spring-1.19.jar	2015/2/11 5:41	Executable Jar File	18 KB
 jettison-1.1.jar	2015/2/11 5:41	Executable Jar File	67 KB
 jsr311-api-1.1.1.jar	2015/2/11 5:41	Executable Jar File	46 KB
 oauth-client-1.19.jar	2015/2/11 5:41	Executable Jar File	15 KB
 oauth-server-1.19.jar	2015/2/11 5:41	Executable Jar File	30 KB
 oauth-signature-1.19.jar	2015/2/11 5:41	Executable Jar File	24 KB

图 4.6 本例所涉及的 Jar 包

对 Kubernetes API 的访问包含如下三个方面。

(1) 指明访问资源的类型。

(2) 访问时的一些选项（参数），比如命名空间、对象的名称、过滤方式（标签和域）、子目录、访问的目标是否是代理和是否用 watch 方式访问等。

(3) 访问的方法，比如增、删、改、查。

在使用 Jersey 框架访问 Kubernetes API 之前，为这三个方面定义了三个对象。第 1 个定义的对象为 ResourceType，它定义了访问资源的类型；第 2 个定义的对象是 Params，它定义了访问 API 时的一些选项，以及通过这些选项如何生成完整的 URI；第 3 个定义的对象是 RestfulClient，它是一个接口，该接口定义了访问 API 的方法（Method）。

ResourceType 是一个 ENUM 类型的对象，定义了 16 种资源，代码如下：

```
package com.hp.k8s.apiclient.imp;
```

```
public enum ResourceType {
    NODES("nodes"),
    NAMESPACES("namespaces"),
    SERVICES("services"),
    REPLICATIONCONTROLLERS("replicationcontrollers"),
    PODS("pods"),
    BINDINGS("bindings"),
    ENDPOINTS("endpoints"),
    SERVICEACCOUNTS("serviceaccounts"),
    SECRETS("secrets"),
    EVENTS("events"),
    COMPONENTSTATUSES("componentstatuses"),
    LIMITRANGES("limitranges"),
    RESOURCEQUOTAS("resourcequotas"),
    PODTEMPLATES("podtemplates"),
    PERSISTENTVOLUMECLAIMS("persistentvolumeclaims"); PERSISTENTVOLUMES
("persistentvolumes");
    private String type;

    private ResourceType(String type) {
        this.type = type;
    }

    public String getType() {
        return type;
    }
}
```

Params 对象的代码如下：

```
package com.hp.k8s.apiclient.imp;

import java.io.UnsupportedEncodingException;
import java.net.URLEncoder;
import java.util.List;
import java.util.Map;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Params {
    private static final Logger LOG = LogManager.getLogger(Params.class.getName());
    private String namespace = null;
    private String name = null;
    private Map<String, String> fields = null;
    private Map<String, String> labels = null;
    private Map<String, String> notLabels = null;
    private Map<String, List<String>> inLabels = null;
```

```
private Map<String, List<String>> notInLabels = null;
private String json = null;
private ResourceType resourceType = null;
private String subPath = null;
private boolean isVisitProxy = false;
private boolean isSetWatcher = false;

public String buildPath() {
    StringBuilder result = (isVisitProxy ? new StringBuilder("/proxy")
        : (isSetWatcher ? new StringBuilder("/watch") : new
StringBuilder(""))));
    if (null != namespace)
        result.append("/namespaces/").append(namespace);

    result.append("/").append(resourceType.getType());
    if (null != name)
        result.append("/").append(name);
    if(null!=subPath)
        result.append("/").append(subPath);

    if (null != labels && !labels.isEmpty() || null != notLabels && !notLabels.
isEmpty())
        || null != inLabels && inLabels.size() > 0 || null != notInLabels
&& notInLabels.size() > 0
        || null != fields && fields.size() > 0) {
        StringBuilder labelSelectorStr = null;
        StringBuilder fieldSelectorStr = null;
        try {
            labelSelectorStr = builderLabelSelector();
            fieldSelectorStr = builderFiledSelector();
        } catch (UnsupportedEncodingException e1) {
            LOG.error(e1);
        }
        if (labelSelectorStr.length() + fieldSelectorStr.length() > 0)
            result.append("?");
        if (labelSelectorStr.length() > 0) {
            result.append("labelSelector=").append(labelSelectorStr.
toString());
        }
        if (fieldSelectorStr.length() > 0) {
            result.append("fieldSelector=").append(fieldSelectorStr.
toString());
        }
    }
}
```

```
        }

    }

    return result.toString();
}

private StringBuilder builderLabelSelector() throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    if (null != labels) {
        for (String key : labels.keySet()) {
            if (result.length() > 0) {
                result.append(",");
            }
            result.append(URLEncoder.encode(key + "=" + labels.get(key),
                    "GBK"));
        }
    }

    if (null != notLabels) {
        for (String key : notLabels.keySet()) {
            if (result.length() > 0) {
                result.append(",");
            }
            result.append(URLEncoder.encode(key + "!=" + labels.get(key),
                    "GBK"));
        }
    }

    if (null != inLabels) {
        for (String key : inLabels.keySet()) {
            if (result.length() > 0) {
                result.append(URLEncoder.encode(",","GBK"));
            }
            result.append(URLEncoder.encode(key + " in (" + listToString(
                    inLabels.get(key), ",") + ")",
                    "GBK"));
        }
    }

    if (null != notInLabels) {
        for (String key : notInLabels.keySet()) {
            if (result.length() > 0) {
                result.append(URLEncoder.encode(",","GBK"));
            }
        }
    }
}
```

```
        result.append(URLEncoder.encode(key + " notin (" + listToString
(inLabels.get(key), ",") + ")", "GBK")));
    }
}

LOG.info("label result: " + result);
return result;
}

private StringBuilder builderFiledSelector() throws UnsupportedEncodingException {
    StringBuilder result = new StringBuilder();
    if (null != fields) {
        for (String key : fields.keySet()) {
            if (result.length() > 0) {
                result.append(",");
            }

            result.append(URLEncoder.encode(key + "=" + fields.get(key),
"GBK"));
        }
    }

    return result;
}

private String listToString(List<String> list, String delim) {
    boolean isFirst = true;
    StringBuilder result = new StringBuilder();
    for (String str : list) {
        if (isFirst) {
            result.append(str);
            isFirst = false;
        } else {
            result.append(delim).append(str);
        }
    }

    return result.toString();
}

public String getNamespace() {
    return namespace;
}

public void setNamespace(String namespace) {
    this.namespace = namespace;
}
```

```
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public Map<String, String> getFields() {
    return fields;
}

public void setFields(Map<String, String> fields) {
    this.fields = fields;
}

public Map<String, String> getLabels() {
    return labels;
}

public void setLabels(Map<String, String> labels) {
    this.labels = labels;
}

public String getJson() {
    return json;
}

public void setJson(String json) {
    this.json = json;
}

public ResourceType getResourceType() {
    return resourceType;
}

public void setResourceType(ResourceType resourceType) {
    this.resourceType = resourceType;
}

public String getSubPath() {
    return subPath;
}

public void setSubPath(String subPath) {
```

```
        this.subPath = subPath;
    }

    public boolean isVisitProxy() {
        return isVisitProxy;
    }

    public void setVisitProxy(boolean isVisitProxy) {
        this.isVisitProxy = isVisitProxy;
    }

    public boolean isSetWatcher() {
        return isSetWatcher;
    }

    public void setSetWatcher(boolean isSetWatcher) {
        this.isSetWatcher = isSetWatcher;
    }

    public Map<String, String> getNotLabels() {
        return notLabels;
    }

    public void setNotLabels(Map<String, String> notLabels) {
        this.notLabels = notLabels;
    }

    public Map<String, List<String>> getInLabels() {
        return inLabels;
    }

    public void setInLabels(Map<String, List<String>> inLabels) {
        this.inLabels = inLabels;
    }

    public Map<String, List<String>> getNotInLabels() {
        return notInLabels;
    }

    public void setNotInLabels(Map<String, List<String>> notInLabels) {
        this.notInLabels = notInLabels;
    }
}
```

Params 对象包含的属性说明如表 4.2 所示。

表 4.2 Params 对象包含的属性列表

属性	说 明
namespace	String 类型属性，指明资源所在的命名空间，如果没有指定该值，则表明访问所有命名空间下的资源对象
name	String 类型属性，在访问单个资源对象时使用，如果没有指定该值，则表明访问该类资源列表
fields	Map<String, String>类型属性，通过资源对象的域值过滤访问结果
labels	Map<String, String>类型属性，通过指定的标签选择器列表来选择资源对象。选择出的资源对象包含标签列表中所列的标签（即 Map 的 key），且所选资源的标签的 value 和标签列表中的 value 值（即 Map 的 value）相等
notLabels	Map<String, String>类型属性，通过指定的标签选择器列表来选择资源对象。选择出的资源对象包含标签列表中所列的标签（即 Map 的 key），且所选资源的标签的 value 和标签列表中的 value 值（即 Map 的 value）不相等
inLabels	Map<String, List<String>>类型属性，通过指定的标签选择器列表来选择资源对象。Map 对象的 key 值为标签名称，Map 对象的 value 值为该标签可能包含的值
notInLabels	Map<String, List<String>>类型属性，通过指定的标签选择器列表来选择资源对象。Map 对象的 key 值为标签名称，Map 对象的 value 值为列表，表明资源对象包含和 key 值同名的标签，且这些标签的值不在该列表中
json	String 类型属性，在创建或修改资源对象时使用，用于向 API Server 提供资源对象的定义
resourceType	ResourceType 类型属性，用于指明访问资源对象的类型
subPath	String 类型属性，用于指明访问资源的子目录
isVisitProxy	Boolean 类型属性，用于指明是否通过 Proxy 的方式访问资源对象
isSetWatcher	Boolean 类型属性，表明是否通过 Watcher 方式访问资源对象

Params 的 buildPath 方法用于构建访问 URL 的完整路径。

接口对象 RestfulClient 定义了访问 API 接口的所有方法（Method），其代码列表如下：

```
package com.hp.k8s.apiclient;

import com.hp.k8s.apiclient.impl.Params;

public interface RestfulClient {
    public String get(Params params); //获得单个资源对象
    public String list(Params params); //获得资源对象列表
    public String create(Params params); //创建资源对象
    public String delete(Params params); //删除某个资源对象
    public String update(Params params); //部分更新某个资源对象
    public String updateWithMediaType(Params params, String mediaType); //通过
mediaType, 实现 Merge
    public String replace(Params params); //替换某个资源对象
    public String options(Params params);
    public String head(Params params);
}
```

其中 get 和 list 方法对应 Kubernetes API 的 GET 方法；create 方法对应 API 中的 POST 方法；delete 方法对应 API 中的 DELETE 方法；update 方法对应 API 中的 PATCH 方法；replace 方法对应 API 中的 PUT 方法；options 方法对应 API 中的 OPTIONS 方法；head 方法对应 API 中的

HEAD 方法。

该接口的基于 Jersey 框架的实现类如下所示：

```
package com.hp.k8s.apiclient.imp;

import javax.ws.rs.core.MediaType;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import com.hp.k8s.apiclient.RestfulClient;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.config.DefaultClientConfig;
import com.sun.jersey.client.urlconnection.URLConnectionClientHandler;

public class JerseyRestfulClient implements RestfulClient {
    private static final Logger LOG = LogManager.getLogger(RestfulClient.
class.getName());
    private static final String METHOD_PATCH = "PATCH";

    private String _baseUrl = null;
    Client _client = null;

    public JerseyRestfulClient(String baseUrl) {
        DefaultClientConfig config = new DefaultClientConfig();
        config.getProperties().put(URLConnectionClientHandler.PROPERTY_HTTP_
URL_CONNECTION_SET_METHOD_WORKAROUND, true);
        _client = Client.create(config);

        this._baseUrl = baseUrl;
    }

    @Override
    public String get(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        String response = resource.accept(MediaType.APPLICATION_JSON_TYPE).
get(String.class);
        LOG.info("Get one resource:\n" + response);

        return response;
    }

    @Override
    public String list(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
```

```

        LOG.info("URL: " + _baseUrl + params.buildPath());
        String response = resource.accept(MediaType.APPLICATION_JSON_TYPE).
get(String.class);

        return response;
    }

    @Override
    public String create(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        LOG.info("URL: " + _baseUrl + params.buildPath());
        LOG.info("Create resource: " + params.getJson());
        String response = (null == params.getJson())
            ? resource.accept(MediaType.APPLICATION_JSON).post(String.class)
            : resource.type(MediaType.APPLICATION_JSON).accept(MediaType.
APPLICATION_JSON).post(String.class,
            params.getJson());

        return response;
    }

    @Override
    public String delete(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        String response = resource.accept(MediaType.APPLICATION_JSON_TYPE).
delete(String.class);
        LOG.info("Delete resource " + params.getResourceType().getType() + "/"
+ params.getName() + " result:\n"
        + response);

        return response;
    }

    @Override
    public String update(Params params) {
        return updateWithMediaType(params, MediaType.APPLICATION_JSON);
    }

    @Override
    public String updateWithMediaType(Params params, String mediaType) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        LOG.info("URL: " + _baseUrl + params.buildPath());
        LOG.info("Patch resource: " + params.getJson());
        String response = resource.type(mediaType).accept(MediaType.APPLICATION_
JSON_TYPE).method(METHOD_PATCH, String.class,
            params.getJson());
        LOG.info("Update resource " + params.buildPath() + " result:\n" +

```

```
response);

        return response;
    }

    @Override
    public String replace(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        LOG.info("URL: " + _baseUrl + params.buildPath());
        LOG.info("Replace resource: " + params.getJson());
        String response = resource.type(MediaType.APPLICATION_JSON_TYPE).accept(
(MediaType.APPLICATION_JSON_TYPE)
            .put(String.class, params.getJson());
        LOG.info("Replace resource " + params.buildPath() + " result:\n" +
response);

        return response;
    }

    @Override
    public String options(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        String response = resource.type(MediaType.APPLICATION_JSON_TYPE).accept(
(MediaType.TEXT_PLAIN_TYPE)
            .options(String.class);
        LOG.info("Get options for resource " + params.getResourceType().getType() +
"/" + params.getName()
            + " result:\n" + response);

        return response;
    }

    @Override
    public String head(Params params) {
        WebResource resource = _client.resource(_baseUrl + params.buildPath());
        String response = resource.accept(MediaType.TEXT_PLAIN_TYPE).head().
getResponseStatus().toString();
        LOG.info("Get head for resource " + params.getResourceType().getType() +
"/" + params.getName() + " result:\n"
            + response);

        return response;
    }

    @Override
    public void close() {
        _client.destroy();
    }
}
```

```

    }
}

}

```

该对象中包含如下代码：

```
config.getProperties().put(URLConnectionClientHandler.PROPERTY_HTTP_URL_CONNECTION_SET_METHOD_WORKAROUND, true);
```

该段代码的作用是使 Jersey 客户端能够支持除标准 REST 方法外的方法，比如 PATCH 方法。该段代码能访问除 watcher 外的所有 Kubernetes API 接口，在后续的章节中我们会举例说明如何访问 Kubernetes API。

4.3.2 Fabric8

Fabric8 包含多款工具包，Kubernetes Client 只是其中之一，也是 Kubernetes 官网上提到的 Java Client API 之一。本例子代码涉及的 Jar 包如图 4.7 所示。

dnsjava-2.1.7.jar	2015/8/31 14:23	Executable Jar File	301 KB
fabric8-utils-2.2.22.jar	2015/8/31 14:23	Executable Jar File	134 KB
jackson-annotations-2.6.0.jar	2015/8/31 16:27	Executable Jar File	46 KB
jackson-core-2.6.1.jar	2015/8/31 16:28	Executable Jar File	253 KB
jackson-databind-2.6.1.jar	2015/8/31 15:56	Executable Jar File	1,140 KB
jackson-dataformat-yaml-2.6.1.jar	2015/8/31 15:56	Executable Jar File	313 KB
jackson-module-jaxb-annotations-2.6.0.jar	2015/8/31 16:24	Executable Jar File	32 KB
json-20141113.jar	2015/8/31 14:23	Executable Jar File	64 KB
kubernetes-api-2.2.22.jar	2015/8/31 14:22	Executable Jar File	72 KB
kubernetes-client-1.3.8.jar	2015/8/31 15:37	Executable Jar File	2,262 KB
kubernetes-model-1.0.12.jar	2015/8/31 15:56	Executable Jar File	2,308 KB
log4j-api-2.3.jar	2015/8/31 16:18	Executable Jar File	133 KB
log4j-core-2.3.jar	2015/8/31 15:56	Executable Jar File	808 KB
log4j-slf4j-impl-2.3.jar	2015/8/31 15:56	Executable Jar File	23 KB
oauth-20100527.jar	2015/8/31 15:56	Executable Jar File	44 KB
openshift-client-1.3.2.jar	2015/8/31 14:23	Executable Jar File	24 KB
slf4j-api-1.7.12.jar	2015/8/31 15:56	Executable Jar File	32 KB
sundr-annotations-0.0.25.jar	2015/8/31 15:56	Executable Jar File	146 KB
validation-api-1.1.0.Final.jar	2015/8/31 14:23	Executable Jar File	63 KB

图 4.7 例子代码涉及的 Jar 包

因为该工具包已经对访问 Kubernetes API 客户端做了较好的封装，因此其访问代码比较简单，其具体的访问过程会在后续的章节举例说明。

Fabric 8 的 Kubernetes API 客户端工具包只能访问 Node、Service、Pod、Endpoints、Events、Namespace、PersistentVolumeclaims、PersistentVolume、ReplicationController、ResourceQuota、Secret 和 ServiceAccount 这几个资源类型，不能使用 OPTIONS 和 HEAD 方法访问资源，且不能以代理方式访问资源，但其对以 watcher 方式访问资源做了很好的支持。

4.3.3 使用说明

首先，举例说明对 API 资源的基本访问，也就是对资源的增、删、改、查，以及替换资源的 status。其中会单独对 Node 和 Pod 的特殊接口做举例说明。表 4.3 列出了各资源对象的基本 API 接口。

表 4.3 各资源对象的基本 API 接口

资源类型	方法	URL Path	说明	备注
NODES	GET	/api/v1/nodes	获取 Node 列表	
	POST	/api/v1/nodes	创建一个 Node 对象	
	DELETE	/api/v1/nodes/{name}	删除一个 Node 对象	
	GET	/api/v1/nodes/{name}	获取一个 Node 对象	
	PATCH	/api/v1/nodes/{name}	部分更新一个 Node 对象	
	PUT	/api/v1/nodes/{name}	替换一个 Node 对象	
NAMESPACES	GET	/api/v1/namespaces	获取 Namespace 列表	
	POST	/api/v1/namespaces	创建一个 Namespace 对象	
	DELETE	/api/v1/namespaces/{name}	删除一个 Namespace 对象	
	GET	/api/v1/namespaces/{name}	获取一个 Namespace 对象	
	PATCH	/api/v1/namespaces/{name}	部分更新一个 Namespace 对象	
	PUT	/api/v1/namespaces/{name}	替换一个 Namespace 对象	
	PUT	/api/v1/namespaces/{name}/finalize	替换一个 Namespace 对象的最终方案对象	在 Fabric8 中没有实现
	PUT	/api/v1/namespaces/{name}/status	替换一个 Namespace 对象的状态	在 Fabric8 中没有实现
SERVICES	GET	/api/v1/services	获取 Service 列表	
	POST	/api/v1/services	创建一个 Service 对象	
	GET	/api/v1/namespaces/{namespace}/services	获取某个 Namespace 下的 Service 列表	
	POST	/api/v1/namespaces/{namespace}/services	在某个 Namespace 下创建列表	
	DELETE	/api/v1/namespaces/{namespace}/services/{name}	删除某个 Namespace 下的一个 Service 对象	
	GET	/api/v1/namespaces/{namespace}/services/{name}	获取某个 Namespace 下的一个 Service 对象	
	PATCH	/api/v1/namespaces/{namespace}/services/{name}	部分更新某个 Namespace 下的一个 Service 对象	
	PUT	/api/v1/namespaces/{namespace}/services/{name}	替换某个 Namespace 下的一个 Service 对象	

续表

资源类型	方法	URL Path	说明	备注
REPLICATIONCONTROLLERS	GET	/api/v1/replicationcontrollers	获取 RC 列表	
	POST	/api/v1/replicationcontrollers	创建一个 RC 对象	
	GET	/api/v1/namespaces/{namespace}/replicationcontrollers	获取某个 Namespace 下的 RC 列表	
	POST	/api/v1/namespaces/{namespace}/replicationcontrollers	在某个 Namespace 下创建一个 RC 对象	
	DELETE	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	删除某个 Namespace 下的 RC 对象	
	GET	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	获取某个 Namespace 下的 RC 对象	
	PATCH	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	部分更新某个 Namespace 下的 RC 对象	
	PUT	/api/v1/namespaces/{namespace}/replicationcontrollers/{name}	替换某个 Namespace 下的 RC 对象	
PODS	GET	/api/v1/pods	获取一个 Pod 列表	
	POST	/api/v1/pods	创建一个 Pod 对象	
	GET	/api/v1/namespaces/{namespace}/pods	获取某个 Namespace 下的 Pod 列表	
	POST	/api/v1/namespaces/{namespace}/pods	在某个 Namespace 下创建一个 Pod 对象	
	DELETE	/api/v1/namespaces/{namespace}/pods/{name}	删除某个 Namespace 下的一个 Pod 对象	
	GET	/api/v1/namespaces/{namespace}/pods/{name}	获取某个 Namespace 下的一个 Pod 对象	
	PATCH	/api/v1/namespaces/{namespace}/pods/{name}	部分更新某个 Namespace 下的一个 Pod 对象	
	PUT	/api/v1/namespaces/{namespace}/pods/{name}	替换某个 Namespace 下的一个 Pod 对象	
	PUT	/api/v1/namespaces/{namespace}/pods/{name}/status	替换某个 Namespace 下的一个 Pod 对象状态	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/binding	创建某个 Namespace 下的一个 Pod 对象的 Binding	在 Fabric8 中没有实现
	GET	/api/v1/namespaces/{namespace}/pods/{name}/exec	连接到某个 Namespace 下的一个 Pod 对象，并执行 exec	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/exec	连接到某个 Namespace 下的一个 Pod 对象，并执行 exec	在 Fabric8 中没有实现

续表

资源类型	方法	URL Path	说明	备注
	GET	/api/v1/namespaces/{namespace}/pods/{name}/log	连接到某个 Namespace 下的一个 Pod 对象，并获取 log 日志信息	在 Fabric8 中没有实现
	GET	/api/v1/namespaces/{namespace}/pods/{name}/portforward	连接到某个 Namespace 下的一个 Pod 对象，并实现端口转发	在 Fabric8 中没有实现
	POST	/api/v1/namespaces/{namespace}/pods/{name}/portforward	连接到某个 Namespace 下的一个 Pod 对象，并实现端口转发	在 Fabric8 中没有实现
BINDINGS	POST	/api/v1/bindings	创建一个 Binding 对象	
	POST	/api/v1/namespaces/{namespace}/bindings	在某个 Namespace 下创建一个 Binding 对象	
ENDPOINTS	GET	/api/v1/endpoints	获取 Endpoint 列表	
	POST	/api/v1/endpoints	创建一个 Endpoint 对象	
	GET	/api/v1/namespaces/{namespace}/endpoints	获取某个 Namespace 下的 Endpoint 对象列表	
	POST	/api/v1/namespaces/{namespace}/endpoints	在某个 Namespace 下创建一个 Endpoint 对象	
	DELETE	/api/v1/namespaces/{namespace}/endpoints/{name}	删除某个 Namespace 下的 Endpoint 对象	
	GET	/api/v1/namespaces/{namespace}/endpoints/{name}	获取某个 Namespace 下的 Endpoint 对象	
	PATCH	/api/v1/namespaces/{namespace}/endpoints/{name}	部分更新某个 Namespace 下的 Endpoint 对象	
	PUT	/api/v1/namespaces/{namespace}/endpoints/{name}	替换某个 Namespace 下的 Endpoint 对象	
SERVICEACCOUNTS	GET	/api/v1/serviceaccounts	获取 Serviceaccount 列表	
	POST	/api/v1/serviceaccounts	创建一个 Serviceaccount 对象	
	GET	/api/v1/namespaces/{namespace}/serviceaccounts	获取某个 Namespace 下的 Serviceaccount 对象列表	
	POST	/api/v1/namespaces/{namespace}/serviceaccounts	在某个 Namespace 下创建一个 Serviceaccount 对象	
	DELETE	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	删除某个 Namespace 下的一个 Serviceaccount 对象	
	GET	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	获取某个 Namespace 下的一个 Serviceaccount 对象	

续表

资源类型	方法	URL Path	说明	备注
	PATCH	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	部分更新某个 Namespace 下的一个 Serviceaccount 对象	
	PUT	/api/v1/namespaces/{namespace}/serviceaccounts/{name}	替换某个 Namespace 下的一个 Serviceaccount 对象	
SECRETS	GET	/api/v1/secrets	获取 Secret 列表	
	POST	/api/v1/secrets	创建一个 Secret 对象	
	GET	/api/v1/namespaces/{namespace}/secrets	获取某个 Namespace 下的 Secret 列表	
	POST	/api/v1/namespaces/{namespace}/secrets	在某个 Namespace 下创建一个 Secret 对象	
	DELETE	/api/v1/namespaces/{namespace}/secrets/{name}	删除某个 Namespace 下的一个 Secret 对象	
	GET	/api/v1/namespaces/{namespace}/secrets/{name}	获取某个 Namespace 下的一个 Secret 对象	
	PATCH	/api/v1/namespaces/{namespace}/secrets/{name}	部分更新某个 Namespace 下的一个 Secret 对象	
	PUT	/api/v1/namespaces/{namespace}/secrets/{name}	替换某个 Namespace 下的一个 Secret 对象	
EVENTS	GET	/api/v1/events	获取 Event 列表	
	POST	/api/v1/events	创建一个 Event 对象	
	GET	/api/v1/namespaces/{namespace}/events	获取某个 Namespace 下的 Event 列表	
	POST	/api/v1/namespaces/{namespace}/events	在某个 Namespace 下创建一个 Event 对象	
	DELETE	/api/v1/namespaces/{namespace}/events/{name}	删除某个 Namespace 下的一个 Event 对象	
	GET	/api/v1/namespaces/{namespace}/events/{name}	获取某个 Namespace 下的一个 Event 对象	
	PATCH	/api/v1/namespaces/{namespace}/events/{name}	部分更新某个 Namespace 下的一个 Event 对象	
	PUT	/api/v1/namespaces/{namespace}/events/{name}	替换某个 Namespace 下的一个 Event 对象	
COMPONENTSTATUSES	GET	/api/v1/componentstatuses	获取 ComponentStatus 列表	
	GET	/api/v1/namespaces/{namespace}/componentstatuses	获取某个 Namespace 下的 Component Status 列表	

续表

资源类型	方法	URL Path	说明	备注
	GET	/api/v1/namespaces/{namespace}/componentstatuses/{name}	获取某个 Namespace 下的一个 ComponentStatus 对象	
LIMITRANGES	GET	/api/v1/limitranges	获取 LimitRange 列表	
	POST	/api/v1/limitranges	创建一个 LimitRange 对象	
	GET	/api/v1/namespaces/{namespace}/limits	获取某个 Namespace 下的 LimitRange 列表	
	POST	/api/v1/namespaces/{namespace}/limits	在某个 Namespace 下创建一个 LimitRange 对象	
	DELETE	/api/v1/namespaces/{namespace}/limits/{name}	删除某个 Namespace 下的一个 LimitRange 对象	
	GET	/api/v1/namespaces/{namespace}/limits/{name}	获取某个 Namespace 下的一个 LimitRange 对象	
	PATCH	/api/v1/namespaces/{namespace}/limits/{name}	部分更新某个 Namespace 下的一个 LimitRange 对象	
	PUT	/api/v1/namespaces/{namespace}/limits/{name}	替换某个 Namespace 下的一个 LimitRange 对象	
RESOURCEQUOTAS	GET	/api/v1/resourcequotas	获取 ResourceQuota 列表	
	POST	/api/v1/resourcequotas	创建一个 ResourceQuota 对象	
	GET	/api/v1/namespaces/{namespace}/resourcequotas	获取某个 Namespace 下的 Resource Quota 列表	
	POST	/api/v1/namespaces/{namespace}/resourcequotas	在某个 Namespace 下创建一个 Resource Quota 对象	
	DELETE	/api/v1/namespaces/{namespace}/resourcequotas/{name}	删除某个 Namespace 下的一个 Resource Quota 对象	
	GET	/api/v1/namespaces/{namespace}/resourcequotas/{name}	获取某个 Namespace 下的一个 Resource Quota 对象	
	PATCH	/api/v1/namespaces/{namespace}/resourcequotas/{name}	部分更新某个 Namespace 下的一个 Resource Quota 对象	
	PUT	/api/v1/namespaces/{namespace}/resourcequotas/{name}	替换某个 Namespace 下的一个 Resource Quota 对象	
	PUT	/api/v1/namespaces/{namespace}/resourcequotas/{name}/status	替换某个 Namespace 下的一个 Resource Quota 对象状态	在 Fabric8 中没有实现

续表

资源类型	方法	URL Path	说明	备注
PODTEMPLATES	GET	/api/v1/podtemplates	获取 PodTemplate 列表	
	POST	/api/v1/podtemplates	创建一个 PodTemplate 对象	
	GET	/api/v1/namespaces/{namespace}/podtemplates	获取某个 Namespace 下的 PodTemplate 列表	
	POST	/api/v1/namespaces/{namespace}/podtemplates	在某个 Namespace 下创建一个 PodTemplate 对象	
	DELETE	/api/v1/namespaces/{namespace}/podtemplates/{name}	删除某个 Namespace 下的一个 PodTemplate 对象	
	GET	/api/v1/namespaces/{namespace}/podtemplates/{name}	获取某个 Namespace 下的一个 PodTemplate 对象	
	PATCH	/api/v1/namespaces/{namespace}/podtemplates/{name}	部分更新某个 Namespace 下的一个 PodTemplate 对象	
	PUT	/api/v1/namespaces/{namespace}/podtemplates/{name}	替换某个 Namespace 下的一个 PodTemplate 对象	
PERSISTENTVOLUMES	GET	/api/v1/persistentvolumes	获取 PersistentVolume 列表	
	POST	/api/v1/persistentvolumes	创建一个 PersistentVolume 对象	
	DELETE	/api/v1/persistentvolumes/{name}	删除一个 PersistentVolume 对象	
	GET	/api/v1/persistentvolumes/{name}	获取一个 PersistentVolume 对象	
	PATCH	/api/v1/persistentvolumes/{name}	部分更新一个 PersistentVolume 对象	
	PUT	/api/v1/persistentvolumes/{name}	替换一个 PersistentVolume 对象	
	PUT	/api/v1/persistentvolumes/{name}/status	替换一个 PersistentVolume 对象状态	在 Fabric8 中没有实现
PERSISTENTVOLUMECLAIMS	GET	/api/v1/persistentvolumeclaims	获取 PersistentVolumeClaim 列表	
	POST	/api/v1/persistentvolumeclaims	创建一个 PersistentVolumeClaim 对象	
	GET	/api/v1/namespaces/{namespace}/persistentvolumeclaims	获取某个 Namespace 下的 PersistentVolumeClaim 列表	
	POST	/api/v1/namespaces/{namespace}/persistentvolumeclaims	在某个 Namespace 下创建一个 PersistentVolumeClaim 对象	
	DELETE	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	删除某个 Namespace 下的一个 PersistentVolumeClaim 对象	
	GET	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	获取某个 Namespace 下的一个 PersistentVolumeClaim 对象	
	PATCH	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	部分更新某个 Namespace 下的一个 PersistentVolumeClaim 对象	

续表

资源类型	方法	URL Path	说明	备注
	PUT	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	替换某个 Namespace 下的一个 Persistent VolumeClaim 对象	
	PUT	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}/status	替换某个 Namespace 下的一个 Persistent VolumeClaim 对象状态	在 Fabric8 中没有实现

首先，举例说明如何通过 API 接口来创建资源对象。我们需要创建访问 API Server 的客户端，基于 Jersey 框架的代码如下：

```
RestfulClient _restfulClient = new JerseyRestfulClient("http://192.168.1.128:8080/api/v1");
```

其中，`http://192.168.1.128:8080` 为 API Server 的地址。基于 Fabric8 框架的代码如下：

```
Config _conf = new Config();
KubernetesClient _kube = new DefaultKubernetesClient("http://192.168.1.128:8080");
```

分别通过上面的两个客户端创建 Namespace 资源对象，基于 Jersey 框架的代码如下：

```
private void testCreateNamespace() {
    Params params = new Params();
    params.setResourceType(ResourceType.NAMESPACES);
    params.setJson(Utils.getJson("namespace.json"));

    LOG.info("Result: " + _restfulClient.create(params));
}
```

其中，“`namespace.json`”为创建 Namespace 资源对象的 JSON 定义，代码如下：

```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "ns-sample"
  }
}
```

基于 Fabric8 框架的代码如下：

```
private void testCreateNamespace() {
    Namespace ns = new Namespace();
    ns.setApiVersion(ApiVersion.V_1);
    ns.setKind("Namespace");
    ObjectMeta om = new ObjectMeta();
    om.setName("ns-fabric8");
    ns.setMetadata(om);

    _kube.namespaces().create(ns);
```

```

    LOG.info(_kube.namespaces().list().getItems().size());
}

```

由于 Fabric8 框架对 Kubernetes API 对象做了很好的封装，对其中的大量对象都做了定义，所以用户可以通过其提供的资源对象去定义 Kubernetes API 对象，例如上面例子中的 Namespace 对象。Fabric8 框架中的 kubernetes-model 工具包用于 API 对象的封装。在上面的例子中，通过 Fabric8 框架提供的类创建了一个名为“ns-fabric8”的命名空间对象。

接下来我们会通过基于 Jeysey 框架的代码去创建两个 Pod 资源对象。在两个例子中，一个是在上面创建的“ns-sample”Namespace 中创建 Pod 资源对象，另一个是为后续创建“cluster service”而创建的 Pod 资源对象。由于基于 Fabric8 框架创建 Pod 资源对象的方法很简单，因此不再用 Fabric8 框架对上述两个例子做说明。通过基于 Jersey 框架创建这两个 Pod 资源对象的代码如下：

```

private void testCreatePod() {
    Params params = new Params();
    params.setResourceType(ResourceType.PODS);
    params.setJson(Utils.getJson("podInNs.json"));
    params.setNamespace("ns-sample");
    LOG.info("Result: " + _restfulClient.create(params));

    params.setJson(Utils.getJson("pod4ClusterService.json"));
    LOG.info("Result: " + _restfulClient.create(params));
}

```

其中，podInNs.json 和 pod4ClusterService.json 是创建两个 Pod 资源对象的定义。podInNs.json 文件的内容如下：

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-sample-in-namespace",
    "namespace": "ns-sample"
  },
  "spec": {
    "containers": [
      {
        "name": "mycontainer",
        "image": "kubeguide/redis-master"
      }
    ]
  }
}
```

pod4ClusterService.json 文件的内容如下：

```
{
  "kind": "Pod",
```

```

"apiVersion": "v1",
"metadata": {
    "name": "pod-sample-4-cluster-service",
    "namespace": "ns-sample",
    "labels": {
        "k8s-cs": "kube-cluster-service",
        "k8s-test": "kube-cluster-test",
        "k8s-sample-app": "kube-service-sample",
        "kkk": "bbb"
    }
},
"spec": {
    "containers": [
        {
            "name": "mycontainer",
            "image": "kubeguide/redis-master"
        }
    ]
}
}

```

下面的例子代码用于获取 Pod 资源列表，其中第 1 部分代码用于获取所有的 Pod 资源对象，第 2、3 部分代码主要是列举如何使用标签选择 Pod 资源对象，最后一部分代码用于举例说明如何使用 field 选择 Pod 资源对象。代码如下：

```

private void testGetPodList() {
    Params params = new Params();
    params.setResourceType(ResourceType.PODS);
    LOG.info("Result: " + _restfulClient.list(params));

    Map<String, String> labels = new HashMap<String, String>();
    labels.put("k8s-cs", "kube-cluster-service");
    labels.put("k8s-sample-app", "kube-service-sample");
    params.setLabels(labels);
    LOG.info("Result: " + _restfulClient.list(params));
    params.setLabels(null);

    Map<String, List<String>> inLabels = new HashMap<String, List<String>>();
    List list = new ArrayList<String>();
    list.add("kube-cluster-service");
    list.add("kube-cluster");
    inLabels.put("k8s-cs", list);
    params.setInLabels(inLabels);
    LOG.info("Result: " + _restfulClient.list(params));
    params.setInLabels(null);

    Map<String, String> fields = new HashMap<String, String>();
    fields.put("metadata.name", "pod-sample-4-cluster-service");
    params.setNamespace("ns-sample");
}

```

```

        params.setFields(fields);
        LOG.info("Result: " + _restfulClient.list(params));
    }
}

```

接下来的例子代码用于替换一个 Pod 对象，在通过 Kubernetes API 替换一个 Pod 资源对象时需要注意两点：

- (1) 在替换该资源对象前，先从 API 中获取该资源对象的 JSON 对象，然后在该 JSON 对象的基础上修改需要替换的部分；

- (2) 在 Kubernetes API 提供的接口中，PUT 方法（replace）只支持替换容器的 image 部分。

代码如下：

```

private void testReplacePod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setJson(Utils.getJson("pod4Replace.json"));
    params.setResourceType(ResourceType.PODS);

    LOG.info("Result: " + _restfulClient.replace(params));
}

```

其中，pod4Replace.json 的内容如下：

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-sample-in-namespace",
    "namespace": "ns-sample",
    "selfLink": "/api/v1/namespaces/ns-sample/pods/pod-sample-in-namespace",
    "uid": "084ff63e-59d3-11e5-8035-000c2921ba71",
    "resourceVersion": "45450",
    "creationTimestamp": "2015-09-13T04:51:01Z"
  },
  "spec": {
    "volumes": [
      {
        "name": "default-token-szoje",
        "secret": {
          "secretName": "default-token-szoje"
        }
      }
    ],
    "containers": [
      {
        "name": "mycontainer",

```

```
"image": "centos",
"resources": {},
"volumeMounts": [
    {
        "name": "default-token-szoje",
        "readOnly": true,
        "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount"
    }
],
"terminationMessagePath": "/dev/termination-log",
"imagePullPolicy": "IfNotPresent"
},
"restartPolicy": "Always",
"dnsPolicy": "ClusterFirst",
"serviceAccountName": "default",
"serviceAccount": "default",
"nodeName": "192.168.1.129"
},
"status": {
    "phase": "Running",
    "conditions": [
        {
            "type": "Ready",
            "status": "True"
        }
    ],
    "hostIP": "192.168.1.129",
    "podIP": "10.1.10.66",
    "startTime": "2015-09-11T15:17:28Z",
    "containerStatuses": [
        {
            "name": "mycontainer",
            "state": {
                "running": {
                    "startedAt": "2015-09-11T15:17:30Z"
                }
            },
            "lastState": {},
            "ready": true,
            "restartCount": 0,
            "image": "kubeguide/redis-master",
            "imageID": "docker://5630952871a38cddffda9ec611f5978ab0933628fc54cd7d7677ce6b17de33f",
            "containerID": "docker://7bf0d454c367418348711556e667fd1ef6a04d7153d24bfcac2e2e06da634a9f"
        }
    ]
}
```

```

        ]
    }
}

```

接下来的两个例子实现了 4.2.4 节中提到的两种 Merge 方式：Merge Patch 和 Strategic Merge Patch。

第 1 种 Merge 方式的示例如下：

```

private void testUpdatePod1() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setJson(Utils.getJson("pod4MergeJsonPatch.json"));
    params.setResourceType(ResourceType.PODS);

    LOG.info("Result: " + _restfulClient.updateWithMediaType(params,
"application/merge-patch+json"));
}

```

其中，pod4MergeJsonPatch.json 的内容如下：

```

{
  "metadata": {
    "labels": {
      "k8s-cs": "kube-cluster-service",
      "k8s-test": "kube-cluster-test",
      "k8s-sa5555mple-app": "kube-service-sample",
      "kkk": "bbb4444"
    }
  }
}

```

第 2 种 Merge 方式（Strategic Merge Patch）的示例如下：

```

private void testUpdatePod2() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setJson(Utils.getJson("pod4StrategicMerge.json"));
    params.setResourceType(ResourceType.PODS);

    LOG.info("Result: " + _restfulClient.updateWithMediaType(params,
"application/strategic-merge-patch+json"));
}

```

其中，pod4StrategicMerge.json 的内容如下：

```

{
  "spec": {

```

```

    "containers": [
        {
            "name": "mycontainer",
            "image": "centos",
            "patchStrategy": "merge",
            "patchMergeKey": "name"
        }
    ]
}
}

```

接下来实现了修改 Pod 资源对象的状态，代码如下：

```

private void testStatusPod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setSubPath("/status");
    params.setJson(Utils.getJson("pod4Status.json"));
    params.setResourceType(ResourceType.PODS);

    _restfulClient.replace(params);
}

```

其中，`pod4Status.json` 的内容如下：

```

{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pod-sample-in-namespace",
    "namespace": "ns-sample",
    "selfLink": "/api/v1/namespaces/ns-sample/pods/pod-sample-in-namespace",
    "uid": "ad1d803f-59ec-11e5-8035-000c2921ba71",
    "resourceVersion": "51640",
    "creationTimestamp": "2015-09-13T07:54:35Z"
  },
  "spec": {
    "volumes": [
      {
        "name": "default-token-szoje",
        "secret": {
          "secretName": "default-token-szoje"
        }
      }
    ],
    "containers": [
      {
        "name": "mycontainer",
        "image": "kubeguide/redis-master",
        "resources": {}
      }
    ]
  }
}

```

```
"volumeMounts": [
    {
        "name": "default-token-szoje",
        "readOnly": true,
        "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount"
    }
],
"terminationMessagePath": "/dev/termination-log",
"imagePullPolicy": "IfNotPresent"
},
],
"restartPolicy": "Always",
"dnsPolicy": "ClusterFirst",
"serviceAccountName": "default",
"serviceAccount": "default",
"nodeName": "192.168.1.129"
},
"status": {
    "phase": "Unknown",
    "conditions": [
        {
            "type": "Ready",
            "status": "false"
        }
    ],
    "hostIP": "192.168.1.129",
    "podIP": "10.1.10.79",
    "startTime": "2015-09-11T18:21:02Z",
    "containerStatuses": [
        {
            "name": "mycontainer",
            "state": {
                "running": {
                    "startedAt": "2015-09-11T18:21:03Z"
                }
            },
            "lastState": {},
            "ready": true,
            "restartCount": 0,
            "image": "kubeguide/redis-master",
            "imageID": "docker://5630952871a38cddffda9ec611f5978ab0933628fcfd54cd
7d7677ce6b17de33f",
            "containerID": "docker://b0e2312643e9a4b59cf1ff5fb7a8468c5777180d5a
8ea5f2f0c9dfddcf3f4cd2"
        }
    ]
}
```

```
}
```

接下来实现了查看 Pod 的 log 日志功能，代码如下：

```
private void testLogPod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setSubPath("/log");
    params.setResourceType(ResourceType.PODS);

    _restfulClient.get(params);
}
```

下面通过 API 访问 Node 的多种接口，代码如下：

```
private void testPoxyNode() {
    Params params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("pods");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("stats");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("spec");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("run/ns-sample/pod/pod-sample-in-namespace");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.NODES);
    _restfulClient.get(params);

    params = new Params();
    params.setName("192.168.1.129");
    params.setSubPath("metrics");
    params.setVisitProxy(true);
```

```

        params.setResourceType(ResourceType.NODES);
        _restfulClient.get(params);
    }
}

```

最后，举例说明如何通过 API 删除资源对象 pod，代码如下：

```

private void testDeletePod() {
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setResourceType(ResourceType.PODS);
    LOG.info("Result: " + _restfulClient.delete(params));
}
}

```

通过 API 接口除了能够对资源对象实现前面列出的基本操作外，还涉及两类特殊接口，一类是 WATCH，一类是 PROXY。这两类特殊接口所包含的接口如表 4.4 所示。

表 4.4 两类特殊接口所包含的接口

资源类型	类别	方法	URL Path	说明
NODES	WATCH	GET	/api/v1/watch/nodes	监听所有节点的变化
		GET	/api/v1/watch/nodes/{name}	监听单个节点的变化
	PROXY	DELETE	/api/v1/proxy/nodes/{name}/{path:*)}	代理 DELETE 请求到节点的某个子目录
		GET	/api/v1/proxy/nodes/{name}/{path:*)}	代理 GET 请求到节点的某个子目录
		HEAD	/api/v1/proxy/nodes/{name}/{path:*)}	代理 HEAD 请求到节点的某个子目录
		OPTIONS	/api/v1/proxy/nodes/{name}/{path:*)}	代理 OPTIONS 请求到节点的某个子目录
		POST	/api/v1/proxy/nodes/{name}/{path:*)}	代理 POST 请求到节点的某个子目录
		PUT	/api/v1/proxy/nodes/{name}/{path:*)}	代理 PUT 请求到节点的某个子目录
		DELETE	/api/v1/proxy/nodes/{name}	代理 DELETE 请求到节点
		GET	/api/v1/proxy/nodes/{name}	代理 GET 请求到节点
		HEAD	/api/v1/proxy/nodes/{name}	代理 HEAD 请求到节点
		OPTIONS	/api/v1/proxy/nodes/{name}	代理 OPTIONS 请求到节点
		POST	/api/v1/proxy/nodes/{name}	代理 POST 请求到节点
		PUT	/api/v1/proxy/nodes/{name}	代理 PUT 请求到节点
SERVICES	WATCH	GET	/api/v1/watch/services	监听所有 Service 的变化
		GET	/api/v1/watch/namespaces/{namespace}/services	监听某个 Namespace 下所有 Service 的变化

续表

资源类型	类别	方法	URL Path	说明
PROXY	PROXY	GET	/api/v1/watch/namespaces/{namespace}/services/{name}	监听某个 Service 的变化
		DELETE	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*)}	代理 DELETE 请求到 Service 的某个子目录
		GET	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*)}	代理 GET 请求到 Service 的某个子目录
		HEAD	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*)}	代理 HEAD 请求到 Service 的某个子目录
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*)}	代理 OPTIONS 请求到 Service 的某个子目录
		POST	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*)}	代理 POST 请求到 Service 的某个子目录
		PUT	/api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*)}	代理 PUT 请求到 Service 的某个子目录
		DELETE	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 DELETE 请求到 Service
		GET	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 GET 请求到 Service
		HEAD	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 HEAD 请求到 Service
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 OPTIONS 请求到 Service
		POST	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 POST 请求到 Service
		PUT	/api/v1/proxy/namespaces/{namespace}/services/{name}	代理 PUT 请求到 Service
REPLICATIONCONTROLLER	WATCH	GET	/api/v1/watch/replicationcontrollers	监听所有 RC 的变化
		GET	/api/v1/watch/namespaces/{namespace}/replicationcontrollers	监听某个 Namespace 下所有 RC 的变化
		GET	/api/v1/watch/namespaces/{namespace}/replicationcontrollers/{name}	监听某个 RC 的变化
PODS	WATCH	GET	/api/v1/watch/pods	监听所有 Pod 的变化
		GET	/api/v1/watch/namespaces/{namespace}/pods	监听某个 Namespace 下所有 Pod 的变化

续表

资源类型	类别	方法	URL Path	说明
PROXY	PROXY	GET	/api/v1/watch/namespaces/{namespace}/pods/{name}	监听某个 Pod 的变化
		DELETE	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*)}	代理 DELETE 请求到 Pod 的某个子目录
		GET	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*)}	代理 GET 请求到 Pod 的某个子目录
		HEAD	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*)}	代理 HEAD 请求到 Pod 的某个子目录
		OPTIONS	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*)}	代理 OPTIONS 请求到 Pod 的某个子目录
		POST	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*)}	代理 POST 请求到 Pod 的某个子目录
		PUT	/api/v1/namespaces/{namespace}/pods/{name}/proxy/{path:*)}	代理 PUT 请求到 Pod 的某个子目录
		DELETE	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 DELETE 请求到 Pod
		GET	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 GET 请求到 Pod
		HEAD	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 HEAD 请求到 Pod
		OPTIONS	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 OPTIONS 请求到 Pod
		POST	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 POST 请求到 Pod
		PUT	/api/v1/namespaces/{namespace}/pods/{name}/proxy	代理 PUT 请求到 Pod
		DELETE	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*)}	代理 DELETE 请求到 Pod 的某个子目录
		GET	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*)}	代理 GET 请求到 Pod 的某个子目录
		HEAD	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*)}	代理 HEAD 请求到 Pod 的某个子目录
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*)}	代理 OPTIONS 请求到 Pod 的某个子目录
		POST	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*)}	代理 POST 请求到 Pod 的某个子目录

续表

资源类型	类别	方法	URL Path	说明
		PUT	/api/v1/proxy/namespaces/{namespace}/pods/{name}/{path:*	代理 PUT 请求到 Pod 的某个子目录
		DELETE	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 DELETE 请求到 Pod
		GET	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 GET 请求到 Pod
		HEAD	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 HEAD 请求到 Pod
		OPTIONS	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 OPTIONS 请求到 Pod
		POST	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 POST 请求到 Pod
		PUT	/api/v1/proxy/namespaces/{namespace}/pods/{name}	代理 PUT 请求到 Pod
<hr/>				
ENDPOINTS	WATCH	GET	/api/v1/watch/endpoints	监听所有 Endpoint 的变化
		GET	/api/v1/watch/namespaces/{namespace}/endpoints	监听某个 Namespace 下所有 Endpoint 的变化
		GET	/api/v1/watch/namespaces/{namespace}/endpoints/{name}	监听某个 Endpoint 的变化
<hr/>				
SERVICEACCOUNT	WATCH	GET	/api/v1/watch/serviceaccounts	监听所有 ServiceAccount 的变化
		GET	/api/v1/watch/namespaces/{namespace}/serviceaccounts	监听某个 Namespace 下所有 ServiceAccount 的变化
		GET	/api/v1/watch/namespaces/{namespace}/serviceaccounts/{name}	监听某个 ServiceAccount 的变化
<hr/>				
SECRET	WATCH	GET	/api/v1/watch/secrets	监听所有 Secret 的变化
		GET	/api/v1/watch/namespaces/{namespace}/secrets	监听某个 Namespace 下所有 Secret 的变化
		GET	/api/v1/watch/namespaces/{namespace}/secrets/{name}	监听某个 Secret 的变化
<hr/>				
EVENTS	WATCH	GET	/api/v1/watch/events	监听所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/events	监听某个 Namespace 下所有 Event 的变化

续表

资源类型	类别	方法	URL Path	说明
		GET	/api/v1/watch/namespaces/{namespace}/events/{name}	监听某个 Event 的变化
LIMITRANGES	WATCH	GET	/api/v1/watch/limitranges	监听所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/limitranges	监听某个 Namespace 下所有 Event 的变化
		GET	/api/v1/watch/namespaces/{namespace}/limitranges/{name}	监听某个 Event 的变化
RESOURCEQUOTAS	WATCH	GET	/api/v1/watch/resourcequotas	监听所有 ResourceQuota 的变化
		GET	/api/v1/watch/namespaces/{namespace}/resourcequotas	监听某个 Namespace 下所有 ResourceQuota 的变化
		GET	/api/v1/watch/namespaces/{namespace}/resourcequotas/{name}	监听某个 ResourceQuota 的变化
PODTEMPLATES	WATCH	GET	/api/v1/watch/podtemplates	监听所有 PodTemplate 的变化
		GET	/api/v1/watch/namespaces/{namespace}/podtemplates	监听某个 Namespace 下所有 PodTemplate 的变化
		GET	/api/v1/watch/namespaces/{namespace}/podtemplates/{name}	监听某个 PodTemplate 的变化
PERSISTENTVOLUMES	WATCH	GET	/api/v1/watch/persistentvolumes	监听所有 PersistentVolume 的变化
		GET	/api/v1/watch/persistentvolumes/{name}	监听某个 PersistentVolume 的变化
PERSISTENTVOLUMECLAIMS	WATCH	GET	/api/v1/watch/persistentvolumeclaims	监听所有 PersistentVolumeClaim 的变化
		GET	/api/v1/watch/namespaces/{namespace}/persistentvolumeclaims	监听某个 Namespace 下所有 PersistentVolumeClaim 的变化
		GET	/api/v1/watch/namespaces/{namespace}/persistentvolumeclaims/{name}	监听某个 PersistentVolumeClaim 的变化

下面基于 Fabric8 实现对资源对象的监听（Watch），代码如下：

```
private void testWatcher() {
    _kube.pods().watch(new io.fabric8.kubernetes.client.Watcher<Pod>() {
        @Override
        public void eventReceived(Action action, Pod pod) {
            System.out.println(action + ":" + pod);
        }
    })
}
```

```
    @Override
    public void onClose(KubernetesClientException e) {
        System.out.println("Closed: " + e);
    }
});
}
```

接下来基于 Jersey 框架实现通过 Proxy 方式访问 Pod。由于 API Server 针对 Pod 资源提供了两种 Proxy 访问接口，所以下面分别用两段代码进行示例说明。代码如下：

```
private void testPoxyPod() {
    //访问第1种 proxy 接口
    Params params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setSubPath("/proxy");
    params.setResourceType(ResourceType.PODS);

    _restfulClient.get(params);

    //访问第2种 proxy 接口
    params = new Params();
    params.setNamespace("ns-sample");
    params.setName("pod-sample-in-namespace");
    params.setVisitProxy(true);
    params.setResourceType(ResourceType.PODS);

    _restfulClient.get(params);
}
```

第 5 章

Kubernetes 运维指南

为了让容器应用在 Kubernetes 集群中运行得更加有效，对 Kubernetes 集群本身也需要进行相应的配置和管理。本章将从 Kubernetes 集群管理、高级案例及 Trouble Shooting 等方面对 Kubernetes 集群的运维和查错进行详细说明，最后对 Kubernetes 1.3 版本开发中的新功能进行介绍。

5.1 Kubernetes 集群管理指南

本节将从 Node 的管理、Label 的管理、Namespace 资源共享、资源配额管理、集群 Master 高可用及集群监控等方面，对 Kubernetes 集群本身的运维管理进行详细说明。

5.1.1 Node 的管理

1. Node 的隔离与恢复

在硬件升级、硬件维护等情况下，我们需要将某些 Node 进行隔离，脱离 Kubernetes 集群的调度范围。Kubernetes 提供了一种机制，既可以将 Node 纳入调度范围，也可以将 Node 脱离调度范围。

创建配置文件 unschedule_node.yaml，在 spec 部分指定 unschedulable 为 true：

```
apiVersion: v1
kind: Node
metadata:
```

```

name: k8s-node-1
labels:
  kubernetes.io/hostname: k8s-node-1
spec:
  unschedulable: true

```

然后，通过 kubectl replace 命令完成对 Node 状态的修改：

```
$ kubectl replace -f unschedule_node.yaml
node "k8s-node-1" replaced
```

查看 Node 的状态，可以观察到在 Node 的状态中增加了一项 SchedulingDisabled：

```
# kubectl get nodes
NAME      STATUS           AGE
k8s-node-1 Ready, SchedulingDisabled 1h
```

对于后续创建的 Pod，系统将不会再向该 Node 进行调度。

也可以不使用配置文件，直接使用 kubectl patch 命令完成：

```
$ kubectl patch node k8s-node-1 -p '{"spec":{"unschedulable":true}}'
```

需要注意的是，将某个 Node 脱离调度范围时，在其上运行的 Pod 并不会自动停止，管理员需要手动停止在该 Node 上运行的 Pod。

同样，如果需要将某个 Node 重新纳入集群调度范围，则将 unschedulable 设置为 false，再次执行 kubectl replace 或 kubectl patch 命令就能恢复系统对该 Node 的调度。

在 Kubernetes 当前的版本中，kubectl 的子命令 cordon 和 uncordon 也用于实现将 Node 进行隔离和恢复调度的操作。

例如，使用 kubectl cordon <node_name> 对某个 Node 进行隔离调度操作：

```
# kubectl cordon k8s-node-1
node "k8s-node-1" cordoned
```

```
# kubectl get nodes
NAME      STATUS           AGE
k8s-node-1 Ready, SchedulingDisabled 1h
```

使用 kubectl uncordon <node_name> 对某个 Node 进行恢复调度操作：

```
# kubectl uncordon k8s-node-1
node "k8s-node-1" uncordoned
```

```
# kubectl get nodes
NAME      STATUS     AGE
k8s-node-1 Ready     1h
```

2. Node 的扩容

在实际生产系统中会经常遇到服务器容量不足的情况，这时就需要购买新的服务器，然后将应用系统进行水平扩展来完成对系统的扩容。

在 Kubernetes 集群中，一个新 Node 的加入是非常简单的。在新的 Node 节点上安装 Docker、kubelet 和 kube-proxy 服务，然后配置 kubelet 和 kube-proxy 的启动参数，将 Master URL 指定为当前 Kubernetes 集群 Master 的地址，最后启动这些服务。通过 kubelet 默认的自动注册机制，新的 Node 将会自动加入现有的 Kubernetes 集群中，如图 5.1 所示。

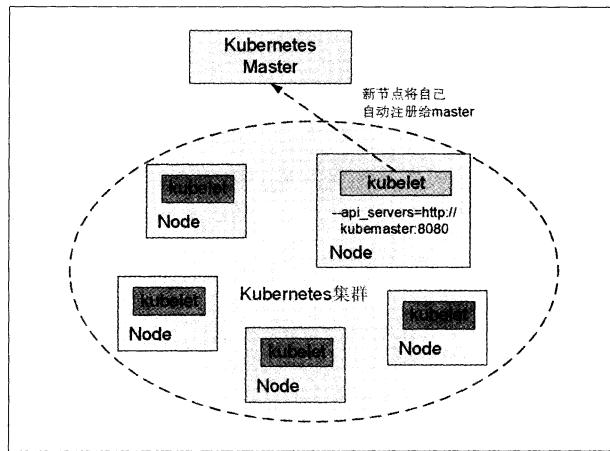


图 5.1 新节点自动注册完成扩容

Kubernetes Master 在接受了新 Node 的注册之后，会自动将其纳入当前集群的调度范围内，在之后创建容器时，就可以向新的 Node 进行调度了。

通过这种机制，Kubernetes 实现了集群中 Node 的扩容。

5.1.2 更新资源对象的 Label

Label（标签）作为用户可灵活定义的对象属性，在正在运行的资源对象上，仍然可以随时通过 kubectl label 命令对其进行增加、修改、删除等操作。

例如，我们要给已创建的 Pod “redis-master-bobr0” 添加一个标签 role=backend：

```
$ kubectl label pod redis-master-bobr0 role=backend
pod "redis-master-bobr0" labeled
```

查看该 Pod 的 Label：

```
$ kubectl get pods -Lrole
```

NAME	READY	STATUS	RESTARTS	AGE	ROLE
redis-master-bobr0	1/1	Running	0	3m	backend

删除一个 Label 时，只需在命令行最后指定 Label 的 key 名并与一个减号相连即可：

```
$ kubectl label pod redis-master-bobr0 role-
pod "redis-master-bobr0" labeled
```

修改一个 Label 的值时，需要加上--overwrite 参数：

```
$ kubectl label pod redis-master-bobr0 role=master --overwrite
pod "redis-master-bobr0" labeled
```

5.1.3 Namespace：集群环境共享与隔离

在一个组织内部，不同的工作组可以在同一个 Kubernetes 集群中工作，Kubernetes 通过命名空间和 Context 的设置来对不同的工作组进行区分，使得它们既可以共享同一个 Kubernetes 集群的服务，也能够互不干扰，如图 5.2 所示。

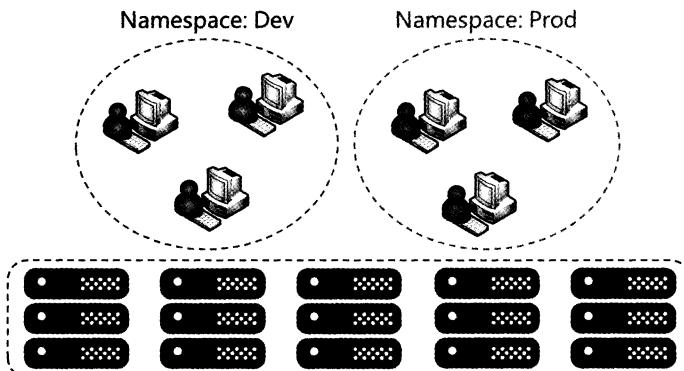


图 5.2 集群环境共享和隔离

假设在我们的组织中有两个工作组：开发组和生产运维组。开发组在 Kubernetes 集群中需要不断创建、修改、删除各种 Pod、RC、Service 等资源对象，以便实现敏捷开发的过程。而生产运维组则需要使用严格的权限设置来确保生产系统中的 Pod、RC、Service 处于正常运行状态且不会被误操作。

1. 创建 namespace

为了在 Kubernetes 集群中实现这两个分组，首先需要创建两个命名空间。

namespace-development.yaml:

```
apiVersion: v1
```

```
kind: Namespace
metadata:
  name: development
```

namespace-production.yaml:

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

使用 `kubectl create` 命令完成命名空间的创建：

```
$ kubectl create -f namespace-development.yaml
namespaces/development
```

```
$ kubectl create -f namespace-production.yaml
namespaces/production
```

查看系统中的命名空间：

```
$ kubectl get namespaces
NAME      LABELS      STATUS
default   <none>     Active
development   name=development   Active
production   name=production   Active
```

2. 定义 Context（运行环境）

接下来，需要为这两个工作组分别定义一个 Context，即运行环境。这个运行环境将属于某个特定的命名空间。

通过 `kubectl config set-context` 命令定义 Context，并将 Context 置于之前创建的命名空间中：

```
$ kubectl config set-cluster kubernetes-cluster --server=https://192.168.1.128:8080
$ kubectl config set-context ctx-dev --namespace=development --cluster=kubernetes-cluster --user=dev
$ kubectl config set-context ctx-prod --namespace=production --cluster=kubernetes-cluster --user=prod
```

使用 `kubectl config view` 命令查看已定义的 Context：

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
  server: http://192.168.1.128:8080
  name: kubernetes-cluster
contexts:
- context:
```

```

cluster: kubernetes-cluster
namespace: development
name: ctx-dev
- context:
  cluster: kubernetes-cluster
  namespace: production
  name: ctx-prod
current-context: ctx-dev
kind: Config
preferences: {}
users: []

```

注意，通过 `kubectl config` 命令在`~/.kube` 目录下生成了一个名为 `config` 的文件，文件内容即以 `kubectl config view` 命令查看到的内容。所以，也可以通过手工编辑该文件的方式来设置 Context。

3. 设置工作组在特定 Context 环境中工作

使用 `kubectl config use-context <context_name>` 命令来设置当前的运行环境。

下面的命令将把当前运行环境设置为“ctx-dev”：

```
$ kubectl config use-context ctx-dev
```

通过这个命令，当前的运行环境即被设置为开发组所需的环境。之后的所有操作都将在名为“development”的命名空间中完成。

现在，以 redis-slave RC 为例创建两个 Pod：

redis-slave-controller.yaml

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  replicas: 2
  selector:
    name: redis-slave
  template:
    metadata:
      labels:
        name: redis-slave
    spec:
      containers:
        - name: slave

```

```
image: kubeguide/guestbook-redis-slave
ports:
- containerPort: 6379

$ kubectl create -f redis-slave-controller.yaml
replicationcontrollers/redis-slave
```

查看创建好的 Pod:

```
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
redis-slave-0feq9  1/1     Running   0          6m
redis-slave-6i0g4  1/1     Running   0          6m
```

可以看到容器被正确创建并运行起来了。而且，由于当前的运行环境是 ctx-dev，所以不会影响到生产运维组的工作。

让我们切换到生产运维组的运行环境:

```
$ kubectl config use-context ctx-prod
```

查看 RC 和 Pod:

```
$ kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR   REPLICAS

$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
```

结果为空，说明看不到开发组创建的 RC 和 Pod。

现在我们为生产运维组也创建两个 redis-slave 的 Pod:

```
$ kubectl create -f redis-slave-controller.yaml
replicationcontrollers/redis-slave
```

查看创建好的 Pod:

```
$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
redis-slave-a4m7s  1/1     Running   0          12s
redis-slave-xyrk  1/1     Running   0          12s
```

可以看到容器被正确创建并运行起来了，并且当前的运行环境是 ctx-prod，也不会影响开发组的工作。

至此，我们为两个工作组分别设置了两个运行环境，在设置好当前的运行环境时，各工作组之间的工作将不会相互干扰，并且它们都能够同一个 Kubernetes 集群中同时工作。

5.1.4 Kubernetes 资源管理

本章从计算资源管理（Compute Resources）、资源配置范围管理（LimitRange）、服务质量管理（QoS）及资源配置额管理（ResourceQuota）等方面，对 Kubernetes 集群内的资源管理进行详细说明，结合实践操作、常见问题分析和一个完整的示例，对 Kubernetes 集群资源管理相关的运维工作提供指导。

1. 计算资源管理（Compute Resources）

在配置 Pod 的时候，我们可以为其中的每个容器指定需要使用的计算资源（CPU 和内存）。

计算资源的配置项分为两种：一种是资源请求（Resource Requests，简称 Requests），表示容器希望被分配到的、可完全保证的资源量，Requests 的值会提供给 Kubernetes 调度器（Kubernetes Scheduler）以便于优化基于资源请求的容器调度；另外一种是资源限制（Resource Limits，简称 Limits），Limits 是容器最多能使用到的资源量的上限，这个上限值会影响节点上发生资源竞争时的解决策略。

当前版本的 Kubernetes 中，计算资源的资源类型分为两种：CPU 和内存（Memory）。这两种资源类型都有一个基本单位：对于 CPU 而言，基本单位是核心数（Cores）；而内存的基本单位是字节数（Bytes）。CPU 和内存一起构成了目前 Kubernetes 中的计算资源（也可简称为资源）。

计算资源是可计量的，能被申请、分配和使用的基础资源，这使之区别于 API 资源（API Resources，例如 Pod 和 services 等）。

1) Pod 和容器的 Requests 和 Limits

Pod 中的每个容器都可以配置以下 4 个参数。

- ◎ spec.container[].resources.requests.cpu。
- ◎ spec.container[].resources.limits.cpu。
- ◎ spec.container[].resources.requests.memory。
- ◎ spec.container[].resources.limits.memory。

这四个参数分别对应容器的 CPU 和内存的 Requests 和 Limits，它们具有以下特点。

- ◎ Requests 和 Limits 都是可选的。在某些集群中如果在 Pod 创建或者更新的时候，没设置资源限制或者资源请求值，那么可能会使用系统提供一个默认值，这个默认值取决于集群的配置。
- ◎ 如果 Request 没有配置，那么默认会被设置为等于 Limits。

◎ 而任何情况下 Limits 都应该设置为大于或者等于 Requests。

以 CPU 为例，图 5.3 显示了未设置 CPU Limits 和设置 CPU Limits 的 CPU 使用率的区别。

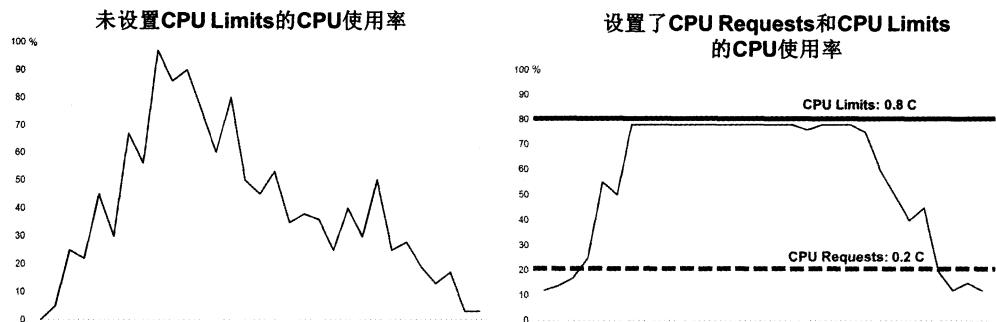


图 5.3 未设置和设置了 CPU Limits 的 CPU 使用率样例

尽管 Requests 和 Limits 只能设置到容器上，但是设置 Pod 级别的 Requests 和 Limits 能极大程度上提高我们对 Pod 管理的便利性和灵活性，因此 Kubernetes 中提供对 Pod 级别的 Requests 和 Limits 配置。对于 CPU 和内存而言，Pod 的 Requests 或 Limits 是指该 Pod 中所有容器的 Requests 或 Limits 的总和（Pod 中没设置 Request 或 Limits 的容器，该项的值被当作 0 或者按照集群配置的默认值来计算）。下面对 CPU 和内存这两种计算资源各自的特点进行说明。

(1) CPU

CPU 的 Requests 和 Limits 是通过 CPU 数 (cpus) 来度量的。CPU 资源值支持最多三位小数：如果一个容器的 spec.container[].resources.requests.cpu 设置为 0.5，那么它会获得半个 CPU；同理如果设置为 1，就会获得 1 个 CPU。0.1CPU 等价于 100m CPU(100 millicpu)，而在 Kubernetes API 中自动将这种小数 0.1 转化为 100m，因此 CPU 的小数最多支持三位数字，而 Kubernetes 官方也更推荐直接使用形如 100m 的 millicpu 作为计量单位。

CPU 资源值是绝对值，而不是相对值：比如 0.1CPU 不管是在单核或者多核机器上都是一样的，都严格等于 0.1 CPU core。

(2) 内存 (Memory)

内存的 Requests 和 Limits 计量单位是字节数 (Bytes)。内存值用使用整数或者定点整数加上国际单位制 (International System of Units) 来表示。国际单位制包括十进制的 E、P、T、G、M、K、m，或二进制的 Ei、Pi、Ti、Gi、Mi、Ki。比如：KiB 与 MiB 是二进制表示的字节单位，而常见的 KB 与 MB 则是十进制表示的字节单位。两种方式的区别举例说明如下：

$$1 \text{ KB (kilobyte)} = 1000 \text{ bytes} = 8000 \text{ bits}$$

$$1 \text{ KiB (kibibyte)} = 2^{10} \text{ bytes} = 1024 \text{ bytes} = 8192 \text{ bits}$$

因此，下面几种内存配置的意思是一样的：128974848、129e6、129M、123Mi

Kubernetes 的计算资源单位是大小写敏感的，因为 m 可以表示千分之一单位（milli unit），而 M 可以表示十进制的 1000，两者的含义不同；同理可知，小写的 k 不是一个合法的资源单位。

以一个 Pod 中的资源配置为例：

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: db
      image: mysql
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
    - name: wp
      image: wordpress
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
```



该 Pod 包含两个容器，每个容器配置的 Requests 都是 0.25 CPU 和 64MiB (2^{26} bytes) 内存，而配置的 Limits 都是 0.5 CPU 和 128MiB (2^{27} bytes) 内存。

这个 Pod 的 Requests 和 Limits 等于 Pod 中所有容器对应配置的总和，所以 Pod 的 Requests 是 0.5 CPU 和 128MiB (2^{27} bytes) 内存，Limits 是 1 CPU 和 256MiB (2^{28} bytes) 内存。

2) 基于 Requests 和 Limits 的 Pod 调度机制

当一个 Pod 创建成功时，Kubernetes 调度器（Scheduler）为该 Pod 选择一个节点（Node）来执行。对于每种计算资源（CPU 和内存）而言，每个节点都有一个能用于运行 Pod 的最大容量值。调度器在调度时，首先要确保调度后该节点上所有 Pod 的 CPU 和内存的 Requests 总和不能超过该节点能提供给 Pod 使用的 CPU 和内存的最大容量值。

例如某个节点上 CPU 资源充足，而内存为 4GB，其中 3GB 可以运行 Pod，某 Pod 的内存

Requests 为 1GB、Limits 为 2GB，那么这个节点上最多可运行 3 个这种 Pod。

这里需要注意的是：可能某些节点上的实际资源使用量非常低，但是如果该节点上已运行 Pod 配置的 Requests 值的总和已经非常高，再加上需要调度的 Pod 的 Requests 值会直接超过该节点提供给 Pod 的资源容量上限，Kubernetes 仍然不会将 Pod 调度到这个节点上。这是因为如果 Kubernetes 将 Pod 调度到该节点上，那么如果后面该节点上运行的 Pod 面临服务峰值等情况，可能会导致 Pod 资源短缺的情况发生。

接着上面的例子，假设该节点已经启动 3 个 Pod 实例，而这 3 个 Pod 的实际内存使用都不足 500MB，那么理论上该节点的可用内存应该大于 1.5GB，但是由于该节点的 Pod Requests 总和已经等于节点的可用内存上限，因此 Kubernetes 不会再将任何 Pod 实例调度到该节点上执行。

3) Requests 和 Limits 资源配置机制

当 kubelet 启动 Pod 的一个容器时，它会将容器的 Requests 和 Limits 值转化为相应的容器启动参数传递给容器执行器（Docker 或者是 rkt）。

如果容器的执行环境是 Docker，那么容器的 4 个参数是这样传递给 Docker 的。

(1) spec.container[].resources.requests.cpu

这个参数会转化为 core 数（比如配置的 100m 会转化为 0.1），然后乘以 1024，再将这个结果作为--cpu-shares 参数的值传递给 docker run 命令。在 docker run 命令中，--cpu-share 参数是一个相对权重值（Relative Weight），这个相对权重值会决定 Docker 在资源竞争时分配给容器的资源比例。举例说明--cpu-shares 参数在 Docker 中的含义：比如两个容器的 CPU Requests 分别设置为 1 和 2，那么容器在 docker run 启动时对应的--cpu-shares 参数值分别为 1024 和 2048，在主机 CPU 资源产生竞争时，Docker 会尝试按照 1 : 2 的配比将 CPU 资源分配给这两个容器使用。

这里需要区分清楚的是：这个参数对于 Kubernetes 而言是绝对值，主要用于 Kubernetes 调度和管理的依据（参见下文 QoS 章节）；同时这个参数值会设置为--cpu-shares 参数传递给 Docker，--cpu-shares 参数对于 Docker 而言又是相对值，主要用于资源分配比例。这两种用途的作用范围不同，所以并不会发生冲突。

(2) spec.container[].resources.limits.cpu

这个参数会转化为 millicore 数（比如配置的 1 会转化为 1000，而配置的 100m 转化为 100），将此值乘以 100000，再除以 1000，然后将结果值作为--cpu-quota 参数的值传递给 docker run 命令。docker run 命令中另外一个参数--cpu-period 默认设置为 100000，表示 Docker 重新计量和分配 CPU 的使用时间间隔为 100000 微秒（100 毫秒）。

Docker 的--cpu-quota 参数和--cpu-period 参数一起配合完成对容器 CPU 的使用限制：比如 Kubernetes 中配置容器的 CPU Limits 为 0.1，那么计算后--cpu-quota 为 10000，而--cpu-period

为 100000，这意味着 Docker 在 100 毫秒内最多给该容器分配 10 毫秒*core 的计算资源用量， $10/100=0.1$ core 的结果与 Kubernetes 配置的意义是一致的。

注意：如果 kubelet 启动参数--cpu-cfs-quota 设置为 true，那么 kubelet 会强制要求所有 Pod 都必须配置 CPU Limits（如果 Pod 没配置，而集群提供了默认配置也可以）。而从 Kubernetes 1.2 版本开始，这个--cpu-cfs-quota 启动参数的默认值就是 true。

(3) spec.container[].resources.requests.memory

这个参数值只提供给 Kubernetes 调度器（Kubernetes Scheduler）作为调度和管理的依据，不会作为任何参数传递给 Docker。

(4) spec.container[].resources.limits.memory

这个参数值会转化为单位为 bytes 的整数，数值会作为--memory 参数传递给 docker run 命令。

如果一个容器在运行过程中使用了超出了其内存 Limits 配置的内存限制值，那么它可能会被“杀掉”，如果这个容器是一个可重启的容器，那么之后它会被 kubelet 重新启动起来。因此容器的 Limits 配置需要进行准确的测试和评估。

与内存 Limits 不同的是 CPU 在容器技术中属于可压缩资源，因此对于 CPU 的 Limits 配置一般不会引发因偶然超标使用而导致容器被系统“杀掉”的情况。

4) 计算资源使用情况监控

Pod 的资源用量会作为 Pod 的状态信息一同上报给 Master。如果集群中配置了 Heapster 来监控集群的性能数据，那么还可以从 Heapster 中查看 Pod 的资源用量信息。

5) 计算资源相关常见问题分析

(1) Pod 状态为 pending，错误信息为 FailedScheduling。

如果 Kubernetes 调度器（Kubernetes Scheduler）在集群中找不到合适的节点来运行 Pod，那么这个 Pod 会一直处于未调度状态，直到调度器找到合适的节点为止。每次调度器尝试调度失败，Kubernetes 都会产生一个事件（event），我们可以通过下面这种方式来查看事件的信息：

```
$ kubectl describe pod frontend | grep -A 3 Events
Events:
  FirstSeen    LastSeen    Count  From            Subobject          PathReason      Message
  36s         5s        6      {scheduler}   FailedScheduling  Failed for reason
PodExceedsFreeCPU and possibly others
```

在上面这个例子中，名为 frontend 的 Pod 由于节点的 CPU 资源不足而调度失败（PodExceedsFreeCPU），同样，如果内存不足也可能导致调度失败（PodExceedsFreeMemory）。

如果一个或者多个 Pod 调度失败且有这类错误，那么我们可以尝试以下几种解决方法。

- ◎ 添加更多的节点到集群中。
- ◎ 停止一些不必要的运行中的 Pod，释放资源。
- ◎ 检查 Pod 的配置，错误的配置可能导致该 Pod 永远都无法被调度执行。比如如果整个集群中所有节点都只有 1 CPU，而 Pod 配置的 CPU Requests 为 2，那么该 Pod 就不会被调度执行。

我们可以使用 `kubectl describe nodes` 命令来查看集群中节点的计算资源容量和已使用量：

```
$ kubectl describe nodes k8s-node-1
Name:           k8s-node-1
...
Capacity:
cpu:            1
memory:        464Mi
pods:          40
Allocated resources (total requests):
cpu:            910m
memory:        2370Mi
pods:          4
...
Pods:           (4 in total)
  Namespace      Name           CPU (milliCPU)
Memory (bytes)
  frontend       webserver-ffj8j      500 (50% of total)
  2097152000 (50% of total)
  kube-system    fluentd-cloud-logging-k8s-node-1   100 (10% of total)
  209715200 (5% of total)
  kube-system    kube-dns-v8-qopgw     310 (31% of total)
  178257920 (4% of total)
TotalResourceLimits:
  CPU(milliCPU):    910 (91% of total)
  Memory(bytes):   2485125120 (59% of total)
...
```

超过可用资源容量上限 (Capacity) 和已分配资源量 (Allocated resources) 差额的 Pod 无法运行在该 Node 上。这个例子中，如果一个 Pod 的 Requests 超过 90 millicpus 或者超过 1341MiB 内存，那么就无法运行在这个节点上。

在后面的资源配置 (Resource Quota) 章节中，我们还可以配置针对一组 Pod 的 Requests 和 Limits 总量的限制，这种限制可以作用于命名空间，通过这种方式我们可以防止一个命名空间下的用户将所有资源全部据为已有。

(2) 容器被强行终止 (Terminated)

如果容器使用的资源超过了它配置的 Limits，那么该容器可能会被强制终止。我们可以通过

过 kubectl describe pod 命令来确认容器是否是因为这个原因被终止的：

```
$ kubectl describe pod simmemleak-hra99
Name:           simmemleak-hra99
Namespace:      default
Image(s):       saadali/simmemleak
Node:           192.168.18.3
Labels:         name=simmemleak
Status:         Running
Reason:
Message:
IP:             172.17.1.3
Replication Controllers:   simmemleak (1/1 replicas created)
Containers:
  simmemleak:
    Image:  saadali/simmemleak
    Limits:
      cpu:        100m
      memory:     50Mi
    State:       Running
    Started:     Tue, 07 Jul 2015 12:54:41 -0700
    Last Termination State: Terminated
    Exit Code:   1
    Started:     Fri, 07 Jul 2015 12:54:30 -0700
    Finished:    Fri, 07 Jul 2015 12:54:33 -0700
    Ready:       False
    Restart Count: 5
Conditions:
  Type  Status
  Ready  False
Events:
  FirstSeen     LastSeen   Count  From          SubobjectPath   Reason     Message
  Tue, 07 Jul 2015 12:53:51 -0700  Tue, 07 Jul 2015 12:53:51 -0700  1  {scheduler }          scheduled
  Successfully assigned simmemleak-hra99 to kubernetes-node-tf0f
  Tue, 07 Jul 2015 12:53:51 -0700  Tue, 07 Jul 2015 12:53:51 -0700  1  {kubelet  kubernetes-node-tf0f}  implicitly required container POD pulled Pod
  container image "gcr.io/google_containers/pause:0.8.0" already present on machine
  Tue, 07 Jul 2015 12:53:51 -0700  Tue, 07 Jul 2015 12:53:51 -0700  1  {kubelet  kubernetes-node-tf0f}  implicitly required container POD created Created
  with docker id 6a41280f516d
  Tue, 07 Jul 2015 12:53:51 -0700  Tue, 07 Jul 2015 12:53:51 -0700  1  {kubelet  kubernetes-node-tf0f}  implicitly required container POD started Started
  with docker id 6a41280f516d
  Tue, 07 Jul 2015 12:53:51 -0700  Tue, 07 Jul 2015 12:53:51 -0700  1  {kubelet  kubernetes-node-tf0f}  spec.containers{simmemleak} created Created
```

with docker id 87348f12526a

Restart Count: 5 说明这个名为 simmemleak 的容器被强制终止并重启了 5 次。

我们可以在使用 kubectl get pod 命令时添加-o go-template=... 格式参数来读取已终止容器之前的状态信息：

```
$ kubectl get pod -o  
go-template='{{range.status.containerStatuses}}{{"Container Name:  
"}}{{.name}}{{"\r\nLastState: "}}{{.lastState}}{{end}}'  
Container Name: simmemleak  
LastState: map[terminated:map[exitCode:137 reason:OOM Killed  
startedAt:2015-07-07T20:58:43Z finishedAt:2015-07-07T20:58:43Z  
containerID:docker://0e4095bba1feccdfe7ef9fb6ebffe972b4b14285d5acdec6f0d3ae8a22f  
ad8b2]]
```

这里我们可以看到这个容器因为 reason:OOM Killed 而被强制终止，说明这个容器的内存超过了限制（Out of Memory）。

6) 计算资源管理的演进

当前版本的 Kubernetes 中的 Requests 和 Limits 都是作用于容器级别的，未来 Kubernetes 计划增加对直接作用于 Pod 级别的资源配置的支持，这种资源配置是能被 Pod 内的所有容器共享的，包括 emptyDir 这种 Pod 级别的 Volume。

从资源的种类来看，目前 Kubernetes 只能支持 CPU 和内存两种计算资源类型，在后续的版本中，Kubernetes 计划支持更多的资源类型，包括节点磁盘空间资源，还将支持自定义的资源类型。

2. 资源的配置范围管理（LimitRange）

默认情况下，Kubernetes 的 Pod 会以无限制的 CPU 和内存运行。这也就意味着 Kubernetes 系统中任何的 Pod 都可以使用其所在节点上的所有可用的 CPU 和内存。通过配置 Pod 的计算资源 Requests 和 Limits，我们可以限制 Pod 的资源使用，但对于 Kubernetes 集群管理员而言，配置每一个 Pod 的 Requests 和 Limits 是烦琐且限制性过强的。更多的时候，我们需要的是对集群内 Request 和 Limits 的配置做一个全局的统一的限制。常见的配置场景如下。

- ◎ 集群中的每个节点有 2GB 内存，集群管理员不希望任何 Pod 申请超过 2GB 的内存：因为整个集群中没有任何节点能满足超过 2GB 内存的请求。如果某个 Pod 的内存配置超过 2GB，那么该 Pod 将永远都无法被调度到任何节点上执行。为了防止这种情况的发生，集群管理员希望能在系统管理功能中设置禁止 Pod 申请超过 2GB 内存。
- ◎ 集群由同一个组织中的两个团队共享，各自分别用来运行生产环境和开发环境。生产环境最多可以使用 8GB 内存，而开发环境最多可以使用 512MB 内存。集群管理员希望通过为这两个环境创建不同的命名空间（namespace）并为每个命名空间设置不同的

限制来满足这个需求。

- 用户创建 Pod 时使用的资源可能会刚好比整个机器资源的上限稍小一点，而恰好剩下的资源大小非常尴尬：不足以运行其他任务但整个集群加起来又非常浪费。因此，集群管理员希望设置每个 Pod 必须至少使用集群平均资源值（CPU 和内存）的 20%，这样集群能够提供更好的资源一致性的调度，从而减少了资源浪费。

针对这些需求，Kubernetes 提供了 LimitRange 机制对 Pod 和容器的 Requests 和 Limits 配置进一步做出限制。在下面的示例中，将说明如何将 LimitRange 应用到一个 Kubernetes 的命名空间（namespace）中，然后说明 LimitRange 的几种限制方式，比如最大及最小范围、Requests 和 Limits 的默认值、Limits 与 Requests 最大比例上限等。

下面通过 LimitRange 的设置和应用对其进行说明。

1) 创建一个 namespace

创建一个名为 limit-example 的 namespace：

```
$ kubectl create namespace limit-example  
namespace "limit-example" created
```

2) 为 namespace 设置 LimitRange

为 namespace “limit-example” 创建一个简单的 LimitRange。创建 limits.yaml 配置文件，内容如下：

```
apiVersion: v1  
kind: LimitRange  
metadata:  
  name: mylimits  
spec:  
  limits:  
  - max:  
      cpu: "4"  
      memory: 2Gi  
    min:  
      cpu: 200m  
      memory: 6Mi  
    maxLimitRequestRatio:  
      cpu: 3  
      memory: 2  
    type: Pod  
  - default:  
      cpu: 300m  
      memory: 200Mi  
    defaultRequest:  
      cpu: 200m
```

```
    memory: 100Mi
  max:
    cpu: "2"
    memory: 1Gi
  min:
    cpu: 100m
    memory: 3Mi
  maxLimitRequestRatio:
    cpu: 5
    memory: 4
  type: Container
```

创建该 LimitRange：

```
$ kubectl create -f limits.yaml --namespace=limit-example
limitrange "mylimits" created
```

查看 namespace limit-example 中的 LimitRange：

```
$ kubectl describe limits mylimits --namespace=limit-example
Name:      mylimits
Namespace: limit-example
Type       Resource     Min      Max     Default Request      Default Limit
Max Limit/Request Ratio
-----  -----
Pod        cpu          200m    4        -          -          3
Pod        memory        6Mi     2Gi     -          -          2
Container   cpu          100m    2        200m      300m      5
Container   memory        3Mi     1Gi     100Mi    200Mi     4
```

下面解释一下 LimitRange 中各项配置的意义和特点。

(1) 不论是 CPU 还是内存，在 LimitRange 中，Pod 和 Container 都可以设置 Min、Max 和 Max Limit/Requests Ratio 这三种参数。Container 还可以设置 Default Request 和 Default Limit 这两种参数，而 Pod 不能设置 Default Request 和 Default Limit。

(2) 对 Pod 和 Container 的五种参数的解释如下。

- ◎ Container 的 Min(上面的 100m 和 3Mi)是 Pod 中所有容器的 Requests 值的下限；Container 的 Max (上面的 2 和 1Gi) 是 Pod 中所有容器的 Limits 值的上限；Container 的 Default Request(上面的 200m 和 100Mi)是 Pod 中所有未指定 Requests 值的容器的默认 Requests 值；Container 的 Default Limit (上面的 300m 和 200Mi) 是 Pod 中所有未指定 Limits 值的容器的默认 Limits 值。对于同一资源类型，这 4 个参数必须满足以下关系： $\text{Min} \leq \text{Default Request} \leq \text{Default Limit} \leq \text{Max}$ 。
- ◎ Pod 的 Min (上面的 200m 和 6Mi) 是 Pod 中所有容器的 Requests 值的总和的下限；Pod

的 Max (上面的 4 和 2Gi) 是 Pod 中所有容器的 Limits 值的总和的上限。当容器未指定 Requests 值或者 Limits 值时, 将使用 Container 的 Default Request 值或者 Default Limit 值。

- ◎ Container 的 Max Limit/Requests Ratio (上面的 5 和 4) 限制了 Pod 中所有容器的 Limits 值与 Requests 值的比例上限; 而 Pod 的 Max Limit/Requests Ratio (上面的 3 和 2) 限制了 Pod 中所有容器的 Limits 值总和与 Requests 值总和的比例上限。

(3) 如果设置了 Container 的 Max, 那么对于该类资源而言, 整个集群中的所有容器都必须设置 Limits, 否则将无法成功创建。Pod 内的容器未配置 Limits 时, 将使用 Default Limit 的值 (本例中的 300m CPU 和 200Mi 内存), 而如果 Default 也未配置则无法成功创建。

(4) 如果设置了 Container 的 Min, 那么对于该类资源而言, 整个进群中的所有容器都必须设置 Requests。如果创建 Pod 的容器时未配置该类资源的 Requests, 那么创建过程会报验证错误。Pod 里容器的 Requests 在未配置时, 可以使用默认值 defaultRequest (本例中的 200m CPU 和 100Mi 内存); 如果未配置而又没有 defaultRequest, 那么会默认等于该容器的 Limits; 如果此时 Limits 也未定义, 那么就会报错。

(5) 对于任意一个 Pod 而言, 该 Pod 中所有容器的 Requests 总和必须大于或等于 6Mi, 而且所有容器的 Limits 总和必须小于或等于 1Gi; 同样, 所有容器的 CPU Requests 总和必须大于或等于 200m, 而且所有容器的 CPU Limits 总和必须小于或等于 2。

(6) Pod 里任何容器的 Limits 与 Requests 的比例不能超过 Container 的 Max Limit/Requests Ratio; Pod 里所有容器的 Limits 总和与 Requests 的总和的比例不能超过 Pod 的 Max Limit/Requests Ratio。

3) 创建 Pod 时触发 LimitRange 限制

最后, 让我们看看 LimitRange 生效时对容器的资源限制效果。

命名空间中的限制 (LimitRange) 只会在 Pod 创建或者更新的时候执行检查。如果手动修改限制 (LimitRange) 为一个新的值, 那么这个新的值不会去检查或限制之前已经在该命名空间中创建好的 Pod。

如果用户创建 Pod 时, 配置的资源值 (CPU 或者内存) 超过了 LimitRange 的限制, 那么该创建过程会报错, 在错误信息中会说明详细的错误原因。

下面通过创建一个单容器 Pod 来展示默认限制是如何配置到 Pod 上的:

```
$ kubectl run nginx --image=nginx --replicas=1 --namespace=limit-example
deployment "nginx" created
```

查看已创建的 Pod:

```
$ kubectl get pods --namespace=limit-example
NAME          READY   STATUS    RESTARTS   AGE
nginx         1/1     Running   0          10s
```

```
nginx-2040093540-s8vzu    1/1      Running   0          11s
```

查看该 Pod 的 resources 相关信息：

```
$ kubectl get pods nginx-2040093540-s8vzu --namespace=limit-example -o yaml | grep resources -C 8
  resourceVersion: "57"
  selfLink: /api/v1/namespaces/limit-example/pods/nginx-2040093540-ivimu
  uid: 67b20741-f53b-11e5-b066-64510658e388
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources:
      limits:
        cpu: 300m
        memory: 200Mi
      requests:
        cpu: 200m
        memory: 100Mi
    terminationMessagePath: /dev/termination-log
    volumeMounts:
```

由于该 Pod 未配置资源 Requests 和 Limits，所以使用了 namespace limit-example 中的默认 CPU 和内存定义的 Requests 和 Limits 值。

下面创建一个超出资源限制的 Pod（使用 3 CPU）：

```
invalid-pod.yaml:
apiVersion: v1
kind: Pod
metadata:
  name: invalid-pod
spec:
  containers:
  - name: kubernetes-serve-hostname
    image: gcr.io/google_containers/serve_hostname
    resources:
      limits:
        cpu: "3"
        memory: 100Mi
```

创建该 Pod，可以看到系统报错了，并且提供了错误原因为超过了限制。

```
$ kubectl create -f invalid-pod.yaml --namespace=limit-example
Error from server: error when creating "invalid-pod.yaml": Pod "invalid-pod" is
forbidden: [Maximum cpu usage per Pod is 2, but limit is 3., Maximum cpu usage per
Container is 2, but limit is 3.]
```

接下来的例子展示了 LimitRange 对 maxLimitRequestRatio 的限制:

```
limit-test-nginx.yaml:
apiVersion: v1
kind: Pod
metadata:
  name: limit-test-nginx
  labels:
    name: limit-test-nginx
spec:
  containers:
  - name: limit-test-nginx
    image: nginx
    resources:
      limits:
        cpu: "1"
        memory: 512Mi
      requests:
        cpu: "0.8"
        memory: 250Mi
```

由于 limit-test-nginx 这个 Pod 的全部内存 Limits 总和与 Requests 总和的比例为 512 : 250, 大于 LimitRange 中定义的 Pod 的内存 maxLimitRequestRatio 值 2, 因此创建会失败:

```
$ kubectl create -f limit-test-nginx.yaml --namespace=limit-example
Error from server: error when creating "limit-test-nginx.yaml": pods
"limit-test-nginx" is forbidden: [memory max limit to request ratio per Pod is 2,
but provided ratio is 2.048000.]
```

下面的例子为满足 LimitRange 限制的 Pod:

```
valid-pod.yaml:
apiVersion: v1
kind: Pod
metadata:
  name: valid-pod
  labels:
    name: valid-pod
spec:
  containers:
  - name: kubernetes-serve-hostname
    image: gcr.io/google_containers/serve_hostname
    resources:
      limits:
        cpu: "1"
        memory: 512Mi
```

创建 Pod 将会成功:

```
$ kubectl create -f valid-pod.yaml --namespace=limit-example
```

```
pod "valid-pod" created
```

查看该 Pod 的资源信息：

```
$ kubectl get pods valid-pod --namespace=limit-example -o yaml | grep -C 6 resources
  uid: 3b1bfd7a-f53c-11e5-b066-64510658e388
spec:
  containers:
    - image: gcr.io/google_containers/serve_hostname
      imagePullPolicy: Always
      name: kubernetes-serve-hostname
      resources:
        limits:
          cpu: "1"
          memory: 512Mi
        requests:
          cpu: "1"
          memory: 512Mi
```

可以看到该 Pod 配置了明确的 Limits 和 Requests，因此该 Pod 不会使用 namespace limit-example 中定义的 default 和 defaultRequest。

需要注意的是，CPU Limits 强制配置这个选项在 Kubernetes 集群中默认是开启的；除非集群管理员在部署 kubelet 时，通过设置参数--cpu-cfs-quota=false 来关闭该限制：

```
$ kubelet --help
Usage of kubelet
...
--cpu-cfs-quota[=true]: Enable CPU CFS quota enforcement for containers that
specify CPU limits
$ kubelet --cpu-cfs-quota=false ...
```

如果集群管理员希望对整个集群中容器或者 Pod 配置的 Requests 和 Limits 做限制，那么可以通过配置 Kubernetes 的命名空间（namespace）上的 LimitRange（资源限制区间）来达到该目的。在 Kubernetes 集群中，如果 Pod 没有显式定义 Limits 和 Requests，那么 Kubernetes 系统会将该 Pod 所在的命名空间中定义的 LimitRange 的 default 和 defaultRequests 配置到该 Pod 上。

3. 资源的服务质量管理 (Resource QoS)

本节对 Kubernetes 如何根据 Pod 的 Requests 和 Limits 配置来实现针对 Pod 的不同级别的资源服务质量控制 (QoS) 进行说明。

在 Kubernetes 的资源 QoS 体系中，需要保证高可靠性的 Pod 可以申请可靠资源，而一些不需要高可靠性的 Pod 可以申请可靠性较低或者不可靠的资源。在计算资源一节中，我们讲到了容器的资源配置分为 Requests 和 Limits，其中 Requests 是 Kubernetes 调度时能为容器提供的完

全可保障的资源量（最低保障），而 Limits 是系统允许容器运行时可能使用到的资源量的上限（最高上限）。Pod 级别的资源配置是通过计算 Pod 内所有容器的资源配置的总和得出来的。

Kubernetes 中 Pod 的 Requests 和 Limits 资源配置有如下特点：如果 Pod 配置的 Requests 值等于 Limits 值，那么该 Pod 可以获得的资源是完全可靠的；而如果 Pod 的 Requests 值小于 Limits 值，那么该 Pod 获得的资源可分成两部分：一部分是完全可靠的资源，资源量大小等于 Requests 值；另外一部分是不可靠的资源，这部分资源最大等于 Limits 与 Requests 的差额值，这份不可靠的资源能够申请到多少，则取决于当时主机上容器可用资源的余量。

通过这种机制，Kubernetes 可以实现节点资源的超售（Over Subscription），比如在 CPU 完全充足的情况下，某机器共有 32GiB 内存可提供给容器使用，容器配置为 Requests 值 1GiB，Limits 值为 2GiB，那么该机器上最多可以同时运行 32 个容器，每个容器最多可使用 2GiB 内存，如果这些容器的内存使用峰值错开，那么所有容器也可以一直正常运行。

超售机制能有效地提高资源的利用率，同时不会影响容器申请的完全可靠资源的可靠性。

1) Requests 和 Limits 对不同计算资源类型的限制机制

根据计算资源章节的内容我们知道，容器的资源配置满足以下两个条件。

- ◎ Requests \leq 节点可用资源。
- ◎ Requests \leq Limits。

Kubernetes 根据 Pod 配置的 Requests 值来调度 Pod，Pod 在成功调度之后会得到 Requests 值定义的资源来运行；而如果 Pod 所在机器上的资源有空余，则 Pod 可以申请更多的资源，最多不能超过 Limits 的值。我们下面看一下 Requests 和 Limits 针对不同计算资源类型的限制机制的差异。这种差异主要取决于计算资源类型是可压缩资源还是不可压缩资源。

(1) 可压缩资源

- ◎ Kubernetes 目前支持的可压缩资源是 CPU。
- ◎ Pod 可以得到 Pod 的 Requests 配置的 CPU 使用量，而是否能使用超过 Requests 值的部分取决于系统的负载和调度。不过由于目前 Kubernetes 和 Docker 的 CPU 隔离机制都是在容器级别隔离的，所以 Pod 级别的资源配置并不能完全得到保障；Pod 级别的 cgroups 正在紧锣密鼓地开发中，如果将来引入，就可以确保 Pod 级别的资源配置准确运行。
- ◎ 空闲 CPU 资源按照容器 Requests 值的比例分配。举例说明：容器 A 的 CPU 配置为 Requests 1 Limits 10，容器 B 的 CPU 配置为 request 2 Limits 8，A 和 B 同时运行在一个节点上，初始状态下容器的可用 CPU 为 3cores，那么 A 和 B 恰好得到它们的 Requests 中定义的 CPU 用量，即 1CPU 和 2CPU。如果 A 和 B 都需要更多的 CPU 资源，而恰

好此时系统的其他任务释放出 1.5CPU，那么这 1.5CPU 将按照 A 和 B 的 Requests 值的比例 1 : 2 分配给 A 和 B，即最终 A 可使用 1.5CPU，B 可使用 3CPU。

- ◎ 如果 Pod 使用了超过 Limits 10 中配置的 CPU 用量，那么 cgroups 会对 Pod 中的容器的 CPU 使用进行限流（throttled）；如果 Pod 没有配置 Limits 10，那么 Pod 会尝试抢占所有空闲的 CPU 资源（Kubernetes 从 1.2 版本开始默认开启--cpu-cfs-quota，因此默认情况下必须配置 Limits）。

（2）不可压缩资源

- ◎ Kubernetes 目前支持的可压缩资源是内存。
- ◎ Pod 可以得到 Requests 中配置的内存。如果 Pod 使用的内存量小于它的 Requests 的配置，那么这个 Pod 可以正常运行（除非出现操作系统级别的内存不足等严重问题）；如果 Pod 使用的内存量超过了它的 Requests 的配置，那么这个 Pod 有可能被 Kubernetes “杀掉”：比如 Pod A 使用了超过 Requests 而不到 Limits 的内存量，此时同一机器上另外一个 Pod B 之前只使用了远少于自己的 Requests 值的内存，而此时程序压力增大，Pod B 向系统申请的总量不超过自己的 Requests 值的内存，那么 Kubernetes 可能会直接杀掉 Pod A；另外一种情况是 Pod A 使用了超过 Requests 而不到 Limits 的内存量，此时 Kubernetes 将一个新的 Pod 调度到这台机器上，新的 Pod 需要使用内存，而只有 Pod A 使用了超过了自己的 Requests 值的内存，那么 Kubernetes 也可能会杀掉 Pod A 来释放内存资源。
- ◎ 如果 Pod 使用的内存量超过了它的 Limits 设置，那么操作系统内核会杀掉 Pod 所有容器的所有进程中使用内存最多的一个，直到内存不超过 Limits 为止。

2) 对调度策略的影响

- ◎ Kubernetes 的 kubelet 通过计算 Pod 中所有容器的 Requests 的总和来决定对 Pod 的调度。
- ◎ 不管是 CPU 还是内存，Kubernetes 调度器和 kubelet 都会确保节点上所有 Pod 的 Requests 的总和不会超过该节点上可分配给容器使用的资源容量上限。

3) 服务质量等级（QoS Classes）

在一个超用（Over Committed，即容器 Limits 总和大于系统容量上限）系统中，由于容器负载的波动可能导致操作系统的资源不足，最终可能会导致部分容器被“杀掉”。在这种情况下，我们当然会希望优先“杀掉”那些不太重要的容器，那么如何衡量重要程度呢？Kubernetes 将容器划分成 3 个 QoS 等级：Guaranteed（完全可靠的）、Burstable（弹性波动、较可靠的）和 Best-Effort（尽力而为、不太可靠的），这三种优先级依次递减，如图 5.4 所示。

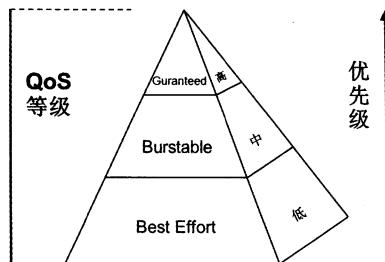


图 5.4 QoS 等级和优先级的关系

从理论上来说，QoS 级别应该作为一个单独的参数来提供 API，并由用户对 Pod 进行配置，这种配置应该与 Requests 和 Limits 无关。但在当前版本的 Kubernetes 的设计中，为了简化模式及避免引入太多的复杂性，QoS 级别直接由 Requests 和 Limits 来定义。在 Kubernetes 中容器的 QoS 级别等于容器所在 Pod 的 QoS 级别，而 Kubernetes 的资源配置定义了 Pod 的三种 QoS 级别，如下所述。

1) Guaranteed（完全可靠的）

如果 Pod 中的所有容器对所有资源类型都定义了 Limits 和 Requests，并且所有容器的 Limits 值都和 Requests 值全部相等（且都不为 0），那么该 Pod 的 QoS 级别就是 Guaranteed。注意：在这种情况下，容器可以不定义 Requests，因为 Requests 值在未定义的时候默认等于 Limits。

下面这两个例子中定义的 Pod QoS 级别就是 Guaranteed：

```
containers:
  name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
  name: bar
  resources:
    limits:
      cpu: 100m
      memory: 100Mi
```

在上面的例子中未定义 Requests 值，所以其默认等于 Limits 值。而下面这个例子中定义的 Requests 和 Limits 的值完全相同：

```
containers:
  name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
```

```
    requests:
      cpu: 10m
      memory: 1Gi
  name: bar
  resources:
    limits:
      cpu: 100m
      memory: 100Mi
    requests:
      cpu: 10m
      memory: 1Gi
```

2) Best-Effort (尽力而为、不太可靠的)

如果 Pod 中所有容器都未定义资源配置 (Requests 和 Limits 都未定义)，那么该 Pod 的 QoS 级别就是 Best-Effort。

例如下面这个 Pod 定义：

```
containers:
  name: foo
  resources:
  name: bar
  resources:
```

3) Burstable (弹性波动、较可靠的)

当一个 Pod 既不是 Guaranteed 级别的，也不是 Best-Effort 级别的时，该 Pod 的 QoS 级别就是 Burstable。Burstable 级别的 Pod 包括两种情况。第 1 种情况是：Pod 中的一部分容器在一种或多种资源类型的资源配置中，定义了 Requests 值和 Limits 值（都不为 0），且 Requests 值小于 Limits 值；第 2 种情况是：Pod 中的一部分容器未定义资源配置 (Requests 和 Limits 都未定义)。注意：容器未定义 Limits 时，Limits 值默认等于节点资源容量上限。

下面几个例子中的 Pod 的 QoS 等级都是 Burstable。

(1) 容器 foo 的 CPU Requests 不等于 Limits:

```
containers:
  name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
    requests:
      cpu: 5m
      memory: 1Gi
  name: bar
  resources:
```

```

limits:
  cpu: 10m
  memory: 1Gi
requests:
  cpu: 10m
  memory: 1Gi

```

(2) 容器 bar 未定义资源配置而容器 foo 定义了资源配置:

```

containers:
  name: foo
  resources:
    limits:
      cpu: 10m
      memory: 1Gi
    requests:
      cpu: 10m
      memory: 1Gi
  name: bar

```

(3) 容器 foo 未定义 CPU, 而容器 bar 未定义内存:

```

containers:
  name: foo
  resources:
    limits:
      memory: 1Gi
  name: bar
  resources:
    limits:
      cpu: 100m

```

(4) 容器 bar 未定义资源配置, 而容器 foo 未定义 Limits 值:

```

containers:
  name: foo
  resources:
    requests:
      cpu: 10m
      memory: 1Gi
  name: bar

```

4) Kubernetes QoS 的工作特点

Pod 的 CPU Requests 无法得到满足(比如节点的系统级任务占用过多的 CPU 导致无法分配给足够的 CPU 给容器使用)时, 容器得到的 CPU 会被压缩限流。

内存由于是不可压缩资源, 所以针对内存资源紧缺的情况, 将按照以下逻辑进行处理。

(1) Best-Effort Pod 的优先级最低, 这类 Pod 中运行的进程会在系统内存紧缺时被第一优先

“杀死”。当然，从另外一个角度来看，Best-Effort Pod 由于没有设置资源 Limits，所以在资源充足的时候，它们可以充分地使用所有的闲置资源。

(2) Burstable Pod 的优先级居中，这类 Pod 初始时会分配较少的可靠资源，但可以按需申请更多的资源。当然，如果整个系统内存紧缺，而又没有 Best-Effort 容器可以被“杀死”以释放资源，则这类 Pod 中的进程可能会被“杀死”。

(3) Guaranteed Pod 的优先级最高，而且一般情况下这类 Pod 只要不超过其资源 Limits 的限制就不会被“杀死”。当然，如果整个系统内存紧缺，而又没有其他更低优先级的容器可以被“杀死”以释放资源，这类 Pod 中的进程也可能被“杀死”。

5) OOM 计分系统

OOM (Out Of Memory) 计分规则包括如下内容。

- ◎ OOM 计分是一个进程消耗内存在系统中占的百分比中不含百分号的数字的值乘以 10 的结果，这个结果是进程 OOM 基础分；将进程 OOM 基础分的分值再加上这个进程的 OOM 分数调整值 OOM_SCORE_ADJ 的值作为进程 OOM 最终分值（除 root 启动的进程外）。在系统发生 OOM 时，OOM Killer 会优先杀掉 OOM 计分更高的进程。
- ◎ 进程的 OOM 计分的基本分数值范围是 0 ~ 1000，如果 A 进程的调整值 OOM_SCORE_ADJ 减去 B 进程的调整值的结果大于 1000，那么 A 进程的 OOM 计分最终值必然大于 B 进程，A 进程会比 B 进程优先被杀死。
- ◎ 不论调整值 OOM_SCORE_ADJ 为多少，任何进程的最终分值范围也是 0 ~ 1000。

在 Kubernetes，不同 QoS 的 OOM 计分调整值规则如表 5.1 所示。

表 5.1 不同 QoS 的 OOM 计分调整值

QoS 等级	oom_score_adj
Guaranteed	-998
BestEffort	1000
Burstable	$\min(\max(2, 1000 - (1000 * \text{memoryRequestBytes}) / \text{machineMemoryCapacityBytes}), 999)$

- ◎ Best-effort Pod 设置 OOM_SCORE_ADJ 调整值为 1000，因此 Best-effort Pod 中的容器里面的所有进程的 OOM 最终分肯定是 1000。
- ◎ Guaranteed Pod 设置 OOM_SCORE_ADJ 调整值为 -998，因此 Guaranteed Pod 中的容器里面的所有进程的 OOM 最终分一般为 0 或者 1（因为基础分不可能为 1000）。
- ◎ Burstable Pod 规则分情况说明：如果 Burstable Pod 的内存 Requests 超过了系统可用内存的 99.8%，那么这个 Pod 的 OOM_SCORE_ADJ 调整值固定为 2；否则，设置

`OOM_SCORE_ADJ` 调整值为 $1000 - 10$ (内存 Requests 占系统可用内存的百分比的无百分号的数字部分的值), 而如果内存 Requests 为 0, 那么 `OOM_SCORE_ADJ` 调整值固定为 999。这样的规则能确保 `OOM_SCORE_ADJ` 调整值的范围为 2~999, 而 `Burstable` Pod 中所有进程的 OOM 最终分数范围为 2~1000。`Burstable` Pod 进程的 OOM 最终分数始终大于 `Guaranteed` Pod 的进程得分, 因此它们会被优先“杀死”。如果一个 `Burstable` Pod 使用的内存比它的内存 Requests 少, 那么可以肯定的是它的所有进程的 OOM 最终分数会小于 1000, 此时能确保它的优先级高于 `Best-effort` Pod。如果一个 `Burstable` Pod 的某个容器中某个进程使用的内存比容器的 `request` 值高, 那么这个进程的 OOM 最终分数会是 1000, 否则它的 OOM 最终分会小于 1000。假设下面容器中有一个占用内存非常大的进程, 那么当一个使用内存超过其 Requests 的 `Burstable` Pod 与另外一个使用内存少于其 Requests 的 `Burstable` Pod 发生内存竞争冲突时, 前者的进程会被系统“杀掉”。如果一个 `Burstable` Pod 内部有多个进程的多个容器发生内存竞争冲突, 那么此时 OOM 评分只能作为参考, 不能保证完全按照资源配置的定义来执行 OOM Kill。

OOM 还有一些特殊的计分规则, 如下所述。

- ◎ `kubelet` 进程和 `Docker` 进程的调整值 `OOM_SCORE_ADJ` 为 -998。
- ◎ 如果配置进程调整值 `OOM_SCORE_ADJ` 为 -999, 那么这类进程不会被 OOM Killer“杀掉”。

6) QoS 的演进

目前 Kubernetes 基于 QoS 的超用机制日趋完善, 但还有一些问题需要解决。

7) 内存 Swap 的支持

当前的 QoS 策略都是假定主机不启用内存 Swap。如果主机启用了 Swap, 那么上面的 QoS 策略可能会失效。举例说明: 两个 `Guaranteed` Pod 都刚好达到了内存 Limits, 那么由于内存 Swap 机制, 它们还可以继续申请使用更多的内存。如果 Swap 空间不足, 那么最终这两个 Pod 中的进程可能会被“杀掉”。由于 Kubernetes 和 Docker 尚不支持内存 Swap 空间的隔离机制, 所以这一功能暂时还未实现。

8) 更丰富的 QoS 策略

当前的 QoS 策略都是基于 Pod 的资源配置 (`Requests` 和 `Limits`) 来定义的, 而资源配置本身又承担着对 Pod 资源管理和限制的功能。两种不同维度的功能使用同一个参数来配置, 可能会导致某些复杂需求无法满足, 比如当前 Kubernetes 无法支持弹性的、高优先级的 Pod。自定义 QoS 优先级能提供更大的灵活性, 完美地实现各类需求, 但同时会引入更高的复杂性, 而且过于灵活的设置会给予用户过高的权限, 对系统管理也提出了更大的挑战。

4. 资源的配额管理（Resource Quotas）

如果一个 Kubernetes 集群被多个用户或者多个团队共享使用，那么就需要考虑共享时对资源公平使用的问题，因为某个用户可能会使用超过基于公平原则分配给其的资源量。

资源配置（Resource Quotas）就是解决这个问题的工具。通过 ResourceQuota 对象，我们可以定义一项资源配置，这个资源配置可以为每一个命名空间（namespace）提供一个总体的资源使用的限制：它可以限制命名空间中某种类型的对象的总数目上限，也可以设置命名空间中 Pod 可以使用到的计算资源的总上限。

典型的资源配置（Resource Quotas）使用方式如下。

- ◎ 不同的团队工作在不同的命名空间下，目前这个是非约束性的，未来版本中可能会通过 ACLs（访问控制列表 Access Control List）的方式来实现强制性约束。
- ◎ 集群管理员为集群中的每个命名空间创建一个或者多个资源配置项。
- ◎ 当用户在命名空间中使用资源（创建 Pod 或者 Service 等）时，Kubernetes 的配额系统会统计、监控和检查资源用量，以确保使用的资源用量没有超过资源配置的配置。
- ◎ 如果创建或者更新应用时，资源使用超过了某项资源配置的限制，那么创建或者更新的请求会报错（HTTP 403 Forbidden），错误信息给出详细的出错原因说明。
- ◎ 如果命名空间中的计算资源（CPU 和内存）的资源配置启用，那么用户必须为相应的资源类型设置 Requests 或 Limits；否则配额系统可能会直接拒绝 Pod 的创建。这里可以使用 LimitRange 机制来为没有配置资源的 Pod 提供默认资源配置。

下面的例子展示了一个非常适合使用资源配置来做资源控制管理的场景。

- ◎ 集群共有 32GB 内存和 16 CPU，两个小组，A 小组使用 20GB 内存和 10 CPU，B 小组使用 10GB 内存和 2 CPU，剩下的 2GB 内存和 2 CPU 作为预留。
- ◎ 在名为 testing 的命名空间中，限制使用 1 CPU 和 1GB 内存；在名为 production 的命名空间中，资源使用不受限制。

在使用资源配置时，需要注意以下两点。

- ◎ 如果集群中总的可用资源小于各命名空间中资源配置的总和，那么可能会导致资源竞争。资源竞争时，Kubernetes 系统使用先到先得的原则。
- ◎ 不管是资源竞争还是配额的修改都不会影响到已经创建的资源使用对象。

1) 在 Master 中开启资源配置选型

资源配置可以通过在 kube-apiserver 的--admission-control=参数值中添加 ResourceQuota 参数进行开启。如果某个命名空间的定义中存在 ResourceQuota，那么对于该命名空间而言，资源配

额就是开启的。一个命名空间可以有多个 ResourceQuota 配置项。

(1) 计算资源配额 (Compute Resource Quota)

资源配置可以限制一个命名空间中所有 Pod 的计算资源的总和。表 5.2 列出了目前 Kubernetes 资源配额支持限制的计算资源类型。

表 5.2 ResourceQuota 的计算资源类型

资源名称	说明
cpu	所有非终止状态的 Pod, CPU Requests 的总和不能超过该值
limits.cpu	所有非终止状态的 Pod, CPU Limits 的总和不能超过该值
limits.memory	所有非终止状态的 Pod, 内存 Limits 的总和不能超过该值
memory	所有非终止状态的 Pod, 内存 Requests 的总和不能超过该值
requests.cpu	所有非终止状态的 Pod, CPU Requests 的总和不能超过该值
requests.memory	所有非终止状态的 Pod, 内存 Requests 的总和不能超过该值

(2) 对象数量配额 (Object Count Quota)

指定类型的对象数量可以被限制。表 5.3 列出了 Kubernetes 资源配额支持限制对象数量的对象类型。

表 5.3 ResourceQuota 的对象类型

资源名称	说明
configmaps	在该命名空间中, 能存在的 ConfigMap 的总数上限
persistentvolumeclaims	在该命名空间中, 能存在的持久卷的总数上限
pods	在该命名空间中, 能存在的非终止状态 Pod 的总数上限。Pod 终止状态等价于 Pod 的 status.phase 状态值为 Failed 或者 Succeed is true
replicationcontrollers	在该命名空间中, 能存在的 RC 的总数上限
resourcequotas	在该命名空间中, 能存在的资源配置项 (ResourcesQuota) 的总数上限
services	在该命名空间中, 能存在的 service 的总数上限
services.loadbalancers	在该命名空间中, 能存在的负载均衡 (LoadBalancer) 的总数上限
services.nodeports	在该命名空间中, 能存在的 NodePort 的总数上限
secrets	在该命名空间中, 能存在的 Secret 的总数上限

例如我们可以通过资源配置来限制命名空间中能创建的 Pod 的最大数量。这种设置可以防止某些用户大量创建 Pod 而迅速耗尽整个集群的 Pod IP 和计算资源。

2) 配额的作用域 (Quota Scopes)

每项资源配置都可以单独配置一组作用域, 配置了作用域的资源配置只会对符合其作用域的资源使用进行计量和限制, 作用域范围内的且超过了资源配置的请求都会报验证错。表 5.4 列出了 ResourceQuota 的 4 种作用域。

表 5.4 ResourceQuota 的作用域

作用域	说明
Terminating	匹配所有 spec.activeDeadlineSeconds >= 0 的 Pod
NotTerminating	匹配所有 spec.activeDeadlineSeconds 是 nil 的 Pod
BestEffort	匹配所有 QoS 为 Best-Effort 的 Pod
NotBestEffort	匹配所有 QoS 不是 Best-Effort 的 Pod

其中，BestEffort 作用域可以限定资源配额来追踪 pods 资源的使用，Terminating、NotTerminating 和 NotBestEffort 这三种作用域可以限定资源配额来追踪以下资源的使用。

- cpu
- limits.cpu
- limits.memory
- memory
- pods
- requests.cpu
- requests.memory

3) 在资源配额 (ResourceQuota) 中设置 Requests 和 Limits

资源配额也可以设置 Requests 和 Limits。

如果资源配额中指定了 requests.cpu 或 requests.memory，那么它会强制要求每一个容器都必须配置自己的 CPU Requests 或 CPU Limits (可使用 LimitRange 提供的默认值)。

同理，如果资源配额中指定了 limits.cpu 或 limits.memory，那么它也会强制要求每一个容器都必须配置自己的内存 Requests 或内存 Limits (可使用 LimitRange 提供的默认值)。

4) 资源配额 (ResourceQuota) 的定义

下面通过几个例子对资源配额进行设置和应用。

与 LimitRange 相似，ResourceQuota 也设置在 namespace 中。创建名为 myspace 的 namespace：

```
$ kubectl create namespace myspace
namespace "myspace" created
```

创建 ResourceQuota 配置文件 compute-resources.yaml，用于设置计算资源的配额：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
```

```

hard:
  pods: "4"
  requests.cpu: "1"
  requests.memory: 1Gi
  limits.cpu: "2"
limits.memory: 2Gi

```

创建该项资源配置:

```
$ kubectl create -f compute-resources.yaml --namespace=myspace
resourcequota "compute-resources" created
```

创建另一个名为 object-counts.yaml 的文件，用于设置对象数量的配额:

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"

```

创建该 ResourceQuota:

```
$ kubectl create -f object-counts.yaml --namespace=myspace
resourcequota "object-counts" created
```

查看各 ResourceQuota 的详细信息:

```
$ kubectl describe quota compute-resources --namespace=myspace
Name:           compute-resources
Namespace:      myspace
Resource        Used Hard
-----  -----
limits.cpu      0    2
limits.memory   0    2Gi
pods            0    4
requests.cpu    0    1
requests.memory 0    1Gi
```

```
$ kubectl describe quota object-counts --namespace=myspace
Name:           object-counts
Namespace:      myspace
Resource        Used Hard
-----  -----
configmaps      0    10
```

persistentvolumeclaims	0	4
replicationcontrollers	0	20
secrets	1	10
services	0	10
services.loadbalancers	0	2

5) 资源配额与集群资源总量的关系

资源配额与集群资源总量是完全独立的。资源配额是通过绝对的单位来配置的：这也就意味着如果集群中新添加了节点，那么资源配额不会自动更新，而该资源配额所对应的命名空间下对象也不能自动地增加资源上限。

在某些情况下，我们可能希望资源配额能支持更复杂的策略，如下所述。

- ◎ 对于不同的租户，按照比例划分整个集群的资源。
- ◎ 允许每个租户都能按照需要来提高资源用量，但是有一个较宽容的限制，以防止意外的资源耗尽情况发生。
- ◎ 探测某个命名空间的需求，添加物理节点并扩大资源配额值。

这些策略可以通过将资源配额作为一个控制模块、手动编写一个控制器（controller）来监控资源使用情况，并调整命名空间上的资源配额的方式来实现。

资源配额将整个集群中的资源总量做了一个静态的划分，但它并没有对集群中的节点（Node）做任何限制：不同命名空间中的 Pod 仍然可以运行到同一个节点上。

5. ResourceQuota 和 LimitRange 实践指南

根据前面对资源管理的介绍，这里将通过一个完整的例子来说明如何通过资源配额和资源配置范围的配合来控制一个命名空间的资源使用。

集群管理员根据集群用户数量来调整集群配置，以达到如下目的：能控制特定命名空间中的资源使用量，最终实现集群的公平使用和成本的控制。

需要实现的功能如下。

- ◎ 限制运行状态的 Pod 的计算资源用量。
- ◎ 限制持久存储卷的数量以控制对存储的访问。
- ◎ 限制负载均衡器的数量以控制成本。
- ◎ 防止滥用网络端口这类稀缺资源。
- ◎ 提供默认的计算资源 Requests 以便于系统做出更优化的调度。

1) 创建命名空间

创建名为 quota-example 的命名空间， namespace.yaml 文件的内容如下：

```
apiVersion: v1
kind: Namespace
metadata:
  name: quota-example

$ kubectl create -f namespace.yaml
namespace "quota-example" created
```

查看命名空间：

```
$ kubectl get namespaces
NAME      STATUS   AGE
default   Active   2m
kube-system   Active   2m
quota-example   Active   39s
```

2) 设置限定对象数目的资源配额

通过设置限定对象的数量的资源配额，可以控制以下资源的数量：

- ◎ 持久存储卷；
- ◎ 负载均衡器；
- ◎ NodePort。

创建名为 object-counts 的 ResourceQuota：

```
object-counts.yaml:
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    persistentvolumeclaims: "2"
    services.loadbalancers: "2"
    services.nodeports: "0"
```

```
$ kubectl create -f object-counts.yaml --namespace=quota-example
resourcequota "object-counts" created
```

配额系统会检测到资源项配额的创建，并且将会统计和限制该命名空间中的资源消耗。

查看该配额是否生效：

```
$ kubectl describe quota object-counts --namespace=quota-example
```

```
Name:          object-counts
Namespace:     quota-example
Resource       Used   Hard
-----
persistentvolumeclaims 0    2
services.loadbalancers 0    2
services.nodeports      0    0
```

至此，配额系统会自动阻止那些使资源用量超过资源配额限定值的请求。

3) 设置限定计算资源的资源配置

下面我们再来创建一项限定计算资源的资源配置，以限制该命名空间中的计算资源的使用总量。

创建名为 compute-resources 的 ResourceQuota:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
  limits.memory: 2Gi

$ kubectl create -f compute-resources.yaml --namespace=quota-example
resourcequota "compute-resources" created
```

查看该配额是否生效:

```
$ kubectl describe quota compute-resources --namespace=quota-example
Name:          compute-resources
Namespace:     quota-example
Resource       Used   Hard
-----
limits.cpu     0    2
limits.memory  0    2Gi
pods          0    4
requests.cpu   0    1
requests.memory 0    1Gi
```

配额系统会自动防止该命名空间下同时拥有超过 4 个非“终止态”的 Pod。此外，由于该项资源配置限制了 CPU 和内存的 Limits 和 Requests 的总量，因此会强制要求该命名空间下的所有容器都必须显示地定义 CPU 和内存的 Limits 和 Requests（可使用默认值，Requests 默认等

于 Limits)。

4) 配置默认 Requests 和 Limits

在命名空间已经配置了限定计算资源的资源配额的情况下，如果尝试在该命名空间下创建一个不指定 Requests 和 Limits 的 Pod，那么 Pod 的创建可能会失败。下面是一个失败的例子。

创建一个 Nginx 的 Deployment:

```
$ kubectl run nginx --image=nginx --replicas=1 --namespace=quota-example
deployment "nginx" created
```

查看创建的 Pod，会发现 Pod 没有创建成功:

```
$ kubectl get pods --namespace=quota-example
```

再查看一下 Deployment 的详细信息:

```
$ kubectl describe deployment nginx --namespace=quota-example
Name:           nginx
Namespace:      quota-example
CreationTimestamp: Mon, 06 Jun 2016 16:11:37 -0400
Labels:          run=nginx
Selector:        run=nginx
Replicas:        0 updated | 1 total | 0 available | 1 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
OldReplicaSets: <none>
NewReplicaSet:  nginx-3137573019 (0/1 replicas created)
...
...
```

本 Deployment 尝试创建一个 Pod，但是失败了，查看其中 ReplicaSet 的详细信息:

```
$ kubectl describe rs nginx-3137573019 --namespace=quota-example
Name:           nginx-3137573019
Namespace:      quota-example
Image(s):       nginx
Selector:        pod-template-hash=3137573019,run=nginx
Labels:          pod-template-hash=3137573019
                 run=nginx
Replicas:        0 current / 1 desired
Pods Status:    0 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen     LastSeen     Count  From                      SubobjectPath  Type
Reason      Message
  -----  -----
  4m          7s            11  {replicaset-controller }          Warning
FailedCreate Error creating: pods "nginx-3137573019-" is forbidden: Failed quota:
```

```
compute-resources: must specify  
limits.cpu,limits.memory,requests.cpu,requests.memory
```

可以看到 Pod 创建失败的原因：Master 拒绝了这个 ReplicaSet 创建 Pod，因为这个 Pod 中没有指定 CPU 和内存的 Requests 和 Limits。

为了避免这种失败，我们可以使用 LimitRange 来为这个命名空间下的所有 Pod 提供一个资源配置的默认值。下面的例子展示了如何为这个命名空间添加一个指定默认资源配置的 LimitRange。

创建一个名为 limits 的 LimitRange：

```
limits.yaml:  
apiVersion: v1  
kind: LimitRange  
metadata:  
  name: limits  
spec:  
  limits:  
    - default:  
        cpu: 200m  
        memory: 512Mi  
    defaultRequest:  
        cpu: 100m  
        memory: 256Mi  
    type: Container
```

```
$ kubectl create -f limits.yaml --namespace=quota-example  
limitrange "limits" created
```

```
$ kubectl describe limits limits --namespace=quota-example  
Name:           limits  
Namespace:      quota-example  
Type           Resource   Min   Max   Default Request  Default Limit  Max Limit/Request  
Ratio  
-----  
-----  
Container memory - - 256Mi          512Mi          -  
Container cpu   - - 100m          200m          -
```

LimitRange 创建成功后，用户在该命名空间下的创建未指定资源配置的 Pod 的请求时，系统会自动为该 Pod 设置默认的资源配置。

例如，每个新建的未指定资源配置的 Pod 都等价于使用下面的资源配置：

```
$ kubectl run nginx \  
--image=nginx \  
--replicas=1 \  
--
```

```
--requests=cpu=100m,memory=256Mi \
--limits=cpu=200m,memory=512Mi \
--namespace=quota-example
```

至此，我们已经为该命名空间配置好了默认的计算资源，我们的 ReplicaSet 应该能够创建 Pod 了。查看一下，创建 Pod 成功了：

```
$ kubectl get pods --namespace=quota-example
NAME           READY   STATUS    RESTARTS   AGE
nginx-3137573019-fvrig  1/1     Running   0          6m
```

接下来，还可以随时查看资源配置的使用情况：

```
$ kubectl describe quota --namespace=quota-example
Name:          compute-resources
Namespace:    quota-example
Resource      Used     Hard
-----
limits.cpu    200m    2
limits.memory 512Mi   2Gi
pods          1        4
requests.cpu  100m    1
requests.memory 256Mi  1Gi
```

```
Name:          object-counts
Namespace:    quota-example
Resource      Used     Hard
-----
persistentvolumeclaims 0      2
services.loadbalancers 0      2
services.nodeports     0      0
```

可以看到每个 Pod 创建时都会消耗掉指定的资源量，而这些使用量都会被 Kubernetes 准确地跟踪、监控和管理。

5) 指定资源配置的作用域

假设我们并不想为某个命名空间配置默认的计算资源配置，而是希望限定在命名空间内运行的 QoS 为 BestEffort 的 Pod 总数，例如将集群中的部分资源用来运行 QoS 为非 BestEffort 的服务，而将闲置的资源用来运行 QoS 为 BestEffort 的服务，即可避免集群的所有资源仅被大量的 BestEffort Pod 耗尽。这可以通过创建两个资源配置（ResourceQuota）来实现。

首先创建一个名为 quota-scopes 的命名空间：

```
$ kubectl create namespace quota-scopes
namespace "quota-scopes" created
```

创建一个名为 best-effort 的 ResourceQuota，指定 Scope 为 BestEffort：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: best-effort
spec:
  hard:
    pods: "10"
  scopes:
    - BestEffort
```

```
$ kubectl create -f best-effort.yaml --namespace=quota-scopes
resourcequota "best-effort" created
```

再创建一个名为 not-best-effort 的 ResourceQuota，指定 Scope 为 NotBestEffort：

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: not-best-effort
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
  scopes:
    - NotBestEffort
```

```
$ kubectl create -f not-best-effort.yaml --namespace=quota-scopes
resourcequota "not-best-effort" created
```

查看创建成功的 ResourceQuota：

```
$ kubectl describe quota --namespace=quota-scopes
Name:      best-effort
Namespace: quota-scopes
Scopes:    BestEffort
* Matches all pods that have best effort quality of service.
Resource   Used   Hard
-----  ----  -----
pods        0      10

Name:      not-best-effort
Namespace: quota-scopes
Scopes:    NotBestEffort
* Matches all pods that do not have best effort quality of service.
Resource   Used   Hard
-----  ----  -----
```

```
limits.cpu      0      2
limits.memory   0      2Gi
pods           0      4
requests.cpu    0      1
requests.memory 0      1Gi
```

之后，对于没有配置 Requests 的 Pod 将会被名为 best-effort 的 ResourceQuota 所限制；而配置了 Requests 的 Pod 会被名为 not-best-effort 的 ResourceQuota 所限制。

创建两个 Deployment：

```
$ kubectl run best-effort-nginx --image=nginx --replicas=8
--namespace=quota-scopes
deployment "best-effort-nginx" created

$ kubectl run not-best-effort-nginx \
--image=nginx \
--replicas=2 \
--requests(cpu=100m, memory=256Mi) \
--limits(cpu=200m, memory=512Mi) \
--namespace=quota-scopes
deployment "not-best-effort-nginx" created
```

名为 best-effort-nginx 的 Deployment 因为没有配置 Requests 和 Limits，所以它的 QoS 级别为 BestEffort，因此它的创建过程由 best-effort 资源配额项来限制，而 not-best-effort 资源配额项不会对它进行限制。best-effort 资源配额项没有限制 Requests 和 Limits，因此 best-effort-nginx Deployment 可以成功地创建 8 个 Pod。

名为 not-best-effort-nginx 的 Deployment 因为配置了 Requests 和 Limits，且二者不相等，所以它的 QoS 级别为 Burstable，因此它的创建过程由 not-best-effort 资源配额项来限制，而 best-effort 资源配额项不会对它进行限制。not-best-effort 资源配额项限制了 Pod 的 Requests 和 Limits 的总上限，not-best-effort-nginx Deployment 并没有超过这个上限，所以可以成功地创建两个 Pod。

查看已经创建的 Pod：

```
$ kubectl get pods --namespace=quota-scopes
NAME                  READY   STATUS    RESTARTS   AGE
best-effort-nginx-3488455095-2qb41   1/1     Running   0          51s
best-effort-nginx-3488455095-3go7n   1/1     Running   0          51s
best-effort-nginx-3488455095-9o2xg   1/1     Running   0          51s
best-effort-nginx-3488455095-eyg40   1/1     Running   0          51s
best-effort-nginx-3488455095-gcs3v   1/1     Running   0          51s
best-effort-nginx-3488455095-rq8p1   1/1     Running   0          51s
best-effort-nginx-3488455095-udhhhd  1/1     Running   0          51s
best-effort-nginx-3488455095-zmk12   1/1     Running   0          51s
not-best-effort-nginx-2204666826-7s161 1/1     Running   0          23s
```

```
not-best-effort-nginx-2204666826-ke746 1/1      Running  0          23s
```

可以看到 10 个 Pod 都创建成功。

再看一下两个资源配置项的使用情况：

```
$ kubectl describe quota --namespace=quota-scopes
Name:           best-effort
Namespace:      quota-scopes
Scopes:         BestEffort
* Matches all pods that have best effort quality of service.
Resource        Used   Hard
-----
pods            8      10
```

```
Name:           not-best-effort
Namespace:      quota-scopes
Scopes:         NotBestEffort
* Matches all pods that do not have best effort quality of service.
Resource        Used   Hard
-----
limits.cpu      400m  2
limits.memory   1Gi   2Gi
pods            2      4
requests.cpu    200m  1
requests.memory 512Mi 1Gi
```

可以看到 best-effort 资源配额项已经统计到了 best-effort-nginx Deployment 中创建的 8 个 Pod 的资源使用信息，而 not-best-effort 资源配额项也统计到了 not-best-effort-nginx Deployment 中创建的两个 Pod 的资源使用信息。

通过这个例子我们可以看到：资源配置的作用域（Scopes）提供了一种将资源集合分割的机制，这种机制使得集群管理员可以更加方便地监控和限制不同类型对象对于各类资源的使用，同时能为资源分配和限制提供更大的灵活度和便利性。

6. 资源管理总结

Kubernetes 中的资源管理的基础是容器和 Pod 的资源配置（Requests 和 Limits）。容器的资源配置（Requests 和 Limits）指定了容器请求的资源和容器能使用的资源上限，而 Pod 的资源配置则是 Pod 中所有容器的资源配置总和的上限。

通过资源配置（Resource Quota）机制，我们可以对命名空间下所有 Pod 使用资源的总量进行限制，也可以对这个命名空间中指定类型的对象的数量进行限制。使用作用域可以让资源配置只对符合特定范围的对象加以限制，因此作用域（Scopes）机制可以使资源配置的策略更加

丰富灵活。

如果我们需要对用户的 Pod 或容器的资源配置做更多的限制，则我们可以使用资源配置范围（LimitRange）来达到这个目的。LimitRange 可以有效地限制 Pod 和容器的资源配置的最大、最小范围，也可以限制 Pod 和容器的 Limits 与 Requests 的最大比例上限，此外 LimitRange 还可以为 Pod 中的容器提供默认的资源配置。

Kubernetes 基于 Pod 的资源配置（Requests 和 Limits）实现了资源服务质量（QoS）。不同 QoS 级别的 Pod 在系统中拥有不同的优先级：高优先级的 Pod 具有更高的可靠性，可以用于运行可靠性要求较高的服务；而低优先级的 Pod 可以实现集群资源的超售，能有效地提高集群资源利用率。

上面的多种机制共同组成了当前版本 Kubernetes 的资源管理体系。这个资源管理体系已经可以满足大部分资源管理的需求了。同时，Kubernetes 资源管理体系仍然在不停地发展和进化中，对于一些目前无法满足的更复杂、更个性化的需求，我们可以继续关注 Kubernetes 未来的发展和变化。

5.1.5 Kubernetes 集群高可用部署方案

Kubernetes 作为容器应用的管理平台，通过对 Pod 的运行状况进行监控，并且根据主机或容器失效的状态将新的 Pod 调度到其他 Node 上，实现了应用层的高可用性。针对 Kubernetes 集群，高可用性还应包含以下两个层面的考虑：etcd 数据存储的高可用性和 Kubernetes Master 组件的高可用性。

1. etcd 高可用部署

etcd 在整个 Kubernetes 集群中处于中心数据库的地位，为保证 Kubernetes 集群的高可用性，首先需要保证数据库不是单故障点。一方面，etcd 需要以集群的方式进行部署，以实现 etcd 数据存储的冗余、备份与高可用性；另一方面，etcd 存储的数据本身也应考虑使用可靠的存储设备。

etcd 集群的部署可以使用静态配置，也可以通过 etcd 提供的 REST API 在运行时动态添加、修改或删除集群中的成员。本节将对 etcd 集群的静态配置进行说明。关于动态修改的操作方法请参考 etcd 官方文档的说明。

首先，规划一个至少 3 台服务器（节点）的 etcd 集群，在每台服务器上安装好 etcd。

部署一个由 3 台服务器组成的 etcd 集群，其配置如表 5.5 所示，其集群部署实例如图 5.5 所示。

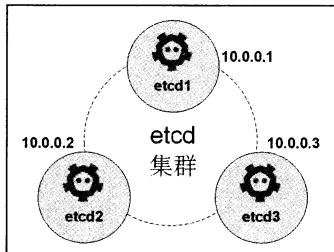


图 5.5 etcd 集群部署实例

表 5.5 etcd 集群的配置

etcd 实例名称	IP 地址
etcd1	10.0.0.1
etcd2	10.0.0.2
etcd3	10.0.0.3

然后修改每台服务器上 etcd 的配置文件 /etc/etcd/etcd.conf。

以 etcd1 为创建集群的实例，需要将其 ETCD_INITIAL_CLUSTER_STATE 设置为“new”。etcd1 的完整配置如下：

```
# [member]
ETCD_NAME=etcd1          #etcd 实例名称
ETCD_DATA_DIR="/var/lib/etcd"    #etcd 数据保存目录
ETCD_LISTEN_CLIENT_URLS="http://10.0.0.1:2379,http://127.0.0.1:2379"  #供外部
客户端使用的 URL
ETCD_ADVERTISE_CLIENT_URLS="http://10.0.0.1:2379,http://127.0.0.1:2379"  #广
播给外部客户端使用的 URL
#[cluster]
ETCD_LISTEN_PEER_URLS="http://10.0.0.1:2380"  #集群内部通信使用的 URL
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.0.0.1:2380"  #广播给集群内其他成员
访问的 URL
ETCD_INITIAL_CLUSTER="etcd1=http://10.0.0.1:2380,etcd2=http://10.0.0.2:2380,
etcd3=http://10.0.0.3:2380"  #初始集群成员列表
ETCD_INITIAL_CLUSTER_STATE="new"  #初始集群状态, new 为新建集群
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"  #集群名称
```

启动 etcd1 服务器上的 etcd 服务：

```
$ systemctl restart etcd
```

启动完成后，就创建了一个名为 etcd-cluster 的集群。

etcd2 和 etcd3 为加入 etcd-cluster 集群的实例，需要将其 ETCD_INITIAL_CLUSTER_STATE 设置为“exist”。etcd2 的完整配置如下（etcd3 的配置略）：

```
# [member]
ETCD_NAME=etcd2          #etcd 实例名称
ETCD_DATA_DIR="/var/lib/etcd"    #etcd 数据保存目录
ETCD_LISTEN_CLIENT_URLS="http://10.0.0.2:2379,http://127.0.0.1:2379"  #供外部
客户端使用的 URL
ETCD_ADVERTISE_CLIENT_URLS="http://10.0.0.2:2379,http://127.0.0.1:2379"  #广
播给外部客户端使用的 URL
```

```

#[cluster]
ETCD_LISTEN_PEER_URLS="http://10.0.0.2:2380"    #集群内部通信使用的 URL
ETCD_INITIAL_ADVERTISE_PEER_URLS="http://10.0.0.2:2380"    #广播给集群内其他成员
使用的 URL
ETCD_INITIAL_CLUSTER="etcd1=http://10.0.0.1:2380,etcd2=http://10.0.0.2:2380,
etcd3=http://10.0.0.3:2380"    #初始集群成员列表
ETCD_INITIAL_CLUSTER_STATE="new"        #初始集群状态, new 为新建集群
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"    #集群名称

```

启动 etcd2 和 etcd3 服务器上的 etcd 服务:

```
$ systemctl restart etcd
```

启动完成后, 在任意 etcd 节点执行 etcdctl cluster-health 命令来查询集群的运行状态:

```

$ etcdctl cluster-health
cluster is healthy
member ce2a822cea30bfca is healthy
member acda82ba1cf790fc is healthy
member eba209cd0012cd2 is healthy

```

在任意 etcd 节点上执行 etcdctl member list 命令来查询集群的成员列表:

```

$ etcdctl member list
ce2a822cea30bfca: name=default peerURLs=http://10.0.0.1:2380,http://127.0.0.1:
7001 clientURLs=http://10.0.0.1:2379,http://127.0.0.1:2379
acda82ba1cf790fc: name=default peerURLs=http://10.0.0.2:2380,http://127.0.0.1:
7001 clientURLs=http://10.0.0.2:2379,http://127.0.0.1:2379
eba209cd40012cd2: name=default peerURLs=http://10.0.0.3:2380,http://127.0.0.1:
7001 clientURLs=http://10.0.0.3:2379,http://127.0.0.1:2379

```

至此, 一个 etcd 集群就创建成功了。

以 kube-apiserver 为例, 将访问 etcd 集群的参数设置为:

```
--etcd-servers=http://10.0.0.1:2379,http://10.0.0.2:2379,http://10.0.0.3:2379
```

在 etcd 集群成功启动之后, 如果需要对集群成员进行修改, 则请参考官方文档的详细说明:

<https://github.com/coreos/etcd/blob/master/Documentation/runtime-configuration.md#cluster-reconfiguration-operations>

对于 etcd 中需要保存的数据的可靠性, 可以考虑使用 RAID 磁盘阵列、高性能存储设备、共享存储文件系统, 或者使用云服务商提供的存储系统等来实现。

2. Master 高可用部署

在 Kubernetes 系统中, Master 服务扮演着总控中心的角色, 主要的三个服务 kube-apiserver、kube-controller-mansger 和 kube-scheduler 通过不断与工作节点上的 kubelet 和 kube-proxy 进行通信来维护整个集群的健康工作状态。如果 Master 的服务无法访问到某个 Node, 则会将该 Node

标记为不可用，不再向其调度新建的 Pod。但对 Master 自身则需要进行额外的监控，使 Master 不成为集群的单故障点，所以对 Master 服务也需要进行高可用方式的部署。

以 Master 的 kube-apiserver、kube-controller-mansger 和 kube-scheduler 三个服务作为一个部署单元，类似于 etcd 集群的典型部署配置。使用至少三台服务器安装 Master 服务，并且需要保证任何时候总有一套 Master 能够正常工作。图 5.6 展示了一种典型的部署方式。

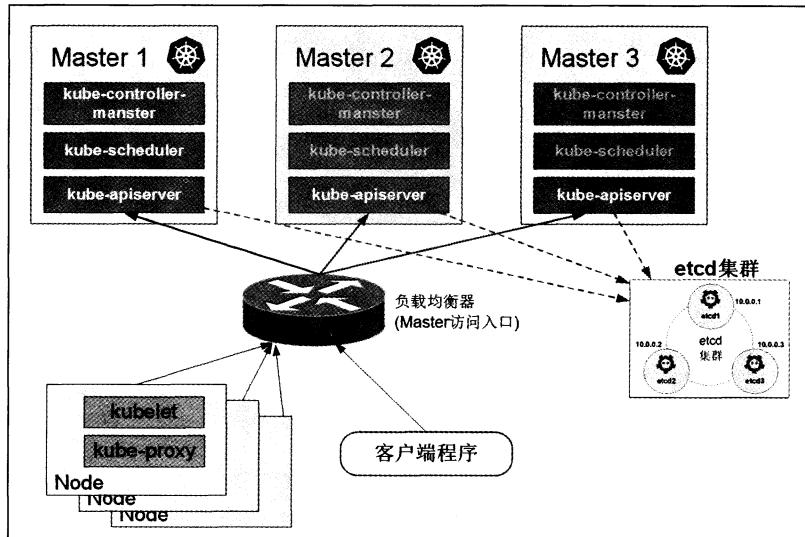


图 5.6 Kubernetes Master 高可用部署架构

Kubernetes 建议 Master 的 3 个组件都以容器的形式启动，启动它们的基础工具是 kubelet，所以它们都将通过 Static Pod 的形式启动并由 kubelet 进行监控和自动重启。而 kubelet 本身的高可用则通过操作系统来完成，例如使用 Linux 的 Systemd 系统进行管理。

注意，如果之前已运行过这 3 个进程，则需要先停止它们，然后启动 kubelet 服务，这 3 个主进程将通过 kubelet 以容器的形式启动和运行。

接下来分别对 kube-apiserver 和 kube-controller-manager、kube-scheduler 的高可用部署进行说明。

1) kube-apiserver 的高可用部署

根据第 2 章的介绍，为 kube-apiserver 预先创建所有需要的 CA 证书和基本鉴权文件等内容，然后在每台服务器上创建其日志文件：

```
# touch /var/log/kube-apiserver.log
```

假设 kubelet 的启动参数指定--config=/etc/kubernetes/manifests，即 Static Pod 定义文件所在的目录，接下来就可以创建 kube-apiserver.yaml 配置文件用于启动 kube-apiserver 了。

```
kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
spec:
  hostNetwork: true
  containers:
    - name: kube-apiserver
      image:
        gcr.io/google_containers/kube-apiserver:9680e782e08a1a1c94c656190011bd02
      command:
        - /bin/sh
        - -c
        - /usr/local/bin/kube-apiserver --etcd-servers=http://127.0.0.1:2379
        --admission-control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota
        --service-cluster-ip-range=169.169.0.0/16 --v=2
        --allow-privileged=False 1>>/var/log/kube-apiserver.log 2>&1
      ports:
        - containerPort: 443
          hostPort: 443
          name: https
        - containerPort: 7080
          hostPort: 7080
          name: http
        - containerPort: 8080
          hostPort: 8080
          name: local
      volumeMounts:
        - mountPath: /srv/kubernetes
          name: srvkube
          readOnly: true
        - mountPath: /var/log/kube-apiserver.log
          name: logfile
        - mountPath: /etc/ssl
          name: etcssl
          readOnly: true
        - mountPath: /usr/share/ssl
          name: usrsharessl
          readOnly: true
        - mountPath: /var/ssl
          name: varssl
          readOnly: true
        - mountPath: /usr/ssl
          name: usrssl
          readOnly: true
```

```
- mountPath: /usr/lib/ssl
  name: usrlibssl
  readOnly: true
- mountPath: /usr/local/openssl
  name: usrlocalopenssl
  readOnly: true
- mountPath: /etc/openssl
  name: etcpopenssl
  readOnly: true
- mountPath: /etc/pki/tls
  name: etcpkitls
  readOnly: true
volumes:
- hostPath:
    path: /srv/kubernetes
  name: srvkube
- hostPath:
    path: /var/log/kube-apiserver.log
  name: logfile
- hostPath:
    path: /etc/ssl
  name: etcssl
- hostPath:
    path: /usr/share/ssl
  name: usrsharessl
- hostPath:
    path: /var/ssl
  name: varssl
- hostPath:
    path: /usr/ssl
  name: usrssl
- hostPath:
    path: /usr/lib/ssl
  name: usrlibssl
- hostPath:
    path: /usr/local/openssl
  name: usrlocalopenssl
- hostPath:
    path: /etc/openssl
  name: etcpopenssl
- hostPath:
    path: /etc/pki/tls
  name: etcpkitls
```

其中，

- ◎ kube-apiserver 需要使用 hostNetwork 模式，即直接使用宿主机网络，以使得客户端能够

通过物理机访问其 API。

- ◎ 镜像的 tag 来源于 kubernetes 发布包中的 kube-apiserver.docker_tag 文件: kubernetes/server/kubernetes-server-linux-amd64/server/bin/kube-apiserver.docker_tag。
- ◎ --etcd-servers: 指定 etcd 服务的 URL 地址。
- ◎ 再加上其他必要的启动参数, 包括--admission-control、--service-cluster-ip-range、CA 证书相关配置等内容。
- ◎ 端口号的设置都配置了 hostPort, 将容器内的端口号直接映射为宿主机的端口号。

将 kube-apiserver.yaml 文件复制到 kubelet 监控的/etc/kubernetes/manifests 目录下, kubelet 将会自动创建 yaml 文件中定义的 kube-apiserver 的 Pod。

接下来在另外两台服务器上重复该操作, 使得每台服务器上都启动一个 kube-apiserver 的 Pod。

2) 为 kube-apiserver 配置负载均衡器

至此, 我们启动了三个 kube-apiserver 实例, 这三个 kube-apiserver 都可以正常工作, 我们需要一个统一的、可靠的、允许部分 Master 节点故障的方式来访问它们, 可以通过部署一个负载均衡器来实现。

在不同的平台下, 负载均衡的实现方式不同: 在一些公用云比如 GCE、AWS、阿里云上都有现成的实现方案; 对于本地集群, 我们可以选择硬件或者软件来实现负载均衡, 比如 Kubernetes 社区推荐的方案 haproxy 和 keepalived 来实现, 其中 haproxy 做负载均衡, 而 keepalived 负责对 haproxy 监控和进行高可用。

在完成 API Server 的负载均衡配置之后, 对其访问还需要注意以下内容。

- ◎ 如果 Master 开启了安全认证机制, 那么需要确保证书中包含负载均衡服务节点的 IP。
- ◎ 对于外部的访问, 比如通过 kubectl 访问 API Server, 那么需要配置为访问 API Server 对应的负载均衡器的 IP 地址。

3) kube-controller-manager 和 kube-scheduler 的高可用配置

不同于 API Server, Master 中另外两个核心组件 kube-controller-manager 和 kube-scheduler 会修改集群的状态信息, 因此对于 kube-controller-manager 和 kube-scheduler 而言, 高可用不仅意味着需要启动多个实例, 还需要这多个实例能实现选举并选举出 leader, 以保证同一时间只有一个实例可以对集群状态信息进行读写, 避免出现同步问题和一致性问题。Kubernetes 对于这种选举机制的实现是采用租赁锁 (lease-lock) 来实现的, 我们可以通过在 kube-controller-manager 和 kube-scheduler 的每个实例的启动参数中设置--leader-elect=true, 来保证同一时间只会运行一个可修改集群信息的实例。

Scheduler 和 Controller Manager 高可用的具体实现方式如下。

首先在每个 Master 节点上创建相应的日志文件：

```
# touch /var/log/kube-scheduler.log  
# touch /var/log/kube-controller-manager.log
```

然后创建 kube-controller-manager 和 kube-scheduler 的 Pod 定义文件：

```
kube-controller-manager.yaml:  
apiVersion: v1  
kind: Pod  
metadata:  
  name: kube-controller-manager  
spec:  
  hostNetwork: true  
  containers:  
    - name: kube-controller-manager  
      image: gcr.io/google_containers/kube-controller-manager:  
fda24638d51a48baa13c35337fc4793  
      command:  
        - /bin/sh  
        - -c  
        - /usr/local/bin/kube-controller-manager --master=127.0.0.1:8080  
          --v=2 --leader-elect=true 1>>/var/log/kube-controller-manager.log 2>&1  
      livenessProbe:  
        httpGet:  
          path: /healthz  
          port: 10252  
        initialDelaySeconds: 15  
        timeoutSeconds: 1  
      volumeMounts:  
        - mountPath: /srv/kubernetes  
          name: srvkube  
          readOnly: true  
        - mountPath: /var/log/kube-controller-manager.log  
          name: logfile  
        - mountPath: /etc/ssl  
          name: etcssl  
          readOnly: true  
        - mountPath: /usr/share/ssl  
          name: usrsharessl  
          readOnly: true  
        - mountPath: /var/ssl  
          name: varssl  
          readOnly: true  
        - mountPath: /usr/ssl  
          name: usrssl
```

```
readOnly: true
- mountPath: /usr/lib/ssl
  name: usrlibssl
  readOnly: true
- mountPath: /usr/local/openssl
  name: usrlocalopenssl
  readOnly: true
- mountPath: /etc/openssl
  name: etcopenssl
  readOnly: true
- mountPath: /etc/pki/tls
  name: etcpkitls
  readOnly: true
volumes:
- hostPath:
    path: /srv/kubernetes
    name: srvkube
- hostPath:
    path: /var/log/kube-controller-manager.log
    name: logfile
- hostPath:
    path: /etc/ssl
    name: etcssl
- hostPath:
    path: /usr/share/ssl
    name: usrsharessl
- hostPath:
    path: /var/ssl
    name: varssl
- hostPath:
    path: /usr/ssl
    name: usrssl
- hostPath:
    path: /usr/lib/ssl
    name: usrlibssl
- hostPath:
    path: /usr/local/openssl
    name: usrlocalopenssl
- hostPath:
    path: /etc/openssl
    name: etcopenssl
- hostPath:
    path: /etc/pki/tls
    name: etcpkitls
```

其中，

- ◎ kube-controller-manager 需要使用 hostNetwork 模式，即直接使用宿主机网络。
- ◎ 镜像的 tag 来源于 kubernetes 发布包中的 kube-controller-manager.docker_tag 文件：kubernetes/server/kubernetes-server-linux-amd64/server/bin/kube-controller-manager.docker_tag。
- ◎ --master：指定 kube-apiserver 服务的 URL 地址。
- ◎ --leader-elect=true：使用 leader 选举机制。

```
kube-scheduler.yaml:  
apiVersion: v1  
kind: Pod  
metadata:  
  name: kube-scheduler  
spec:  
  hostNetwork: true  
  containers:  
    - name: kube-scheduler  
      image:  
        gcr.io/google_containers/kube-scheduler:34d0b8f8b31e27937327961528739bc9  
        command:  
          - /bin/sh  
          - -c  
          - /usr/local/bin/kube-scheduler --master=127.0.0.1:8080 --v=2  
        --leader-elect=true 1>>/var/log/kube-scheduler.log 2>&1  
      livenessProbe:  
        httpGet:  
          path: /healthz  
          port: 10251  
        initialDelaySeconds: 15  
        timeoutSeconds: 1  
      volumeMounts:  
        - mountPath: /var/log/kube-scheduler.log  
          name: logfile  
        - mountPath: /var/run/secrets/kubernetes.io/serviceaccount  
          name: default-token-s8ejd  
          readOnly: true  
      volumes:  
        - hostPath:  
            path: /var/log/kube-scheduler.log  
            name: logfile
```

其中，

- ◎ kube-scheduler 需要使用 hostNetwork 模式，即直接使用宿主机网络。

- 镜像的 tag 来源于 kubernetes 发布包中的 kube-scheduler.docker_tag 文件:kubernetes/server/kubernetes-server-linux-amd64/server/bin/kube-scheduler.docker_tag。
- --master: 指定 kube-apiserver 服务的 URL 地址。
- --leader-elect=true: 使用 leader 选举机制。

将这两个 yaml 文件复制到 kubelet 监控的/etc/kubernetes/manifests 目录下, kubelet 将会自动创建 yaml 文件中定义的 kube-controller-manager 和 kube-scheduler 的 Pod。

至此, 我们完成了 Kubernetes Master 组件高可用的完整配置, 配合 etcd 存储的高可用, 整个 Kubernetes 集群的高可用已经全部完成。最后, 只需要确认集群中所有访问 API Server 的地方都已经将访问地址修改为负载均衡的地址, 就可以保证集群高可用的正常工作了。

3. Master 高可用架构的演进

在当前的版本中, kubelet 可以设置“--api-servers”启动参数来指定多个 kube-apiserver, 但是当第 1 个 kube-apiserver 不可用之后, kubelet 无法连接到后面的 kube-apiserver, 也就是说只有第 1 个 kube-apiserver 起作用。如果这个问题得到解决, 则 kubelet 无须通过额外的负载均衡器就能连接到多个 API Server 了。

另外, 除了 kubelet, 其他核心组件 kube-controller-manager、kube-scheduler 和 kube-proxy 都需要配置 kube-apiserver, 目前它们的启动参数“--master”仅支持配置一个 kube-apiserver, 还无法支持多个 kube-apiserver 的配置。

Kubernetes 计划在后续的版本中支持多个 Master 的配置, 实现不需要负载均衡器的 Master 高可用架构。

5.1.6 Kubernetes 集群监控

1. 通过 cAdvisor 页面查看容器的运行状态

开源软件 cAdvisor (Container Advisor) 是用于监控容器运行状态的利器之一 (cAdvisor 项目的主页为 <https://github.com/google/cadvisor>), 它被用于多个与 Docker 相关的开源项目中。

在 Kubernetes 系统中, cAdvisor 已被默认集成到了 kubelet 组件内, 当 kubelet 服务启动时, 它会自动启动 cAdvisor 服务, 然后 cAdvisor 会实时采集所在节点的性能指标及在节点上运行的容器的性能指标。kubelet 的启动参数--cadvisor-port 可自定义 cAdvisor 对外提供服务的端口号, 默认为 4194。

cAdvisor 提供了 Web 页面可供浏览器访问。例如 Kubernetes 集群中的一个 Node 的 IP 地址

是192.168.18.3，则在浏览器中输入网址`http://192.168.18.3:4194`来访问cAdvisor的监控页面。cAdvisor的主页显示了主机的实时运行状态，包括CPU使用情况、内存使用情况、网络吞吐量及文件系统使用情况等信息。

图5.7展示了cAdvisor的几个性能监控页面。

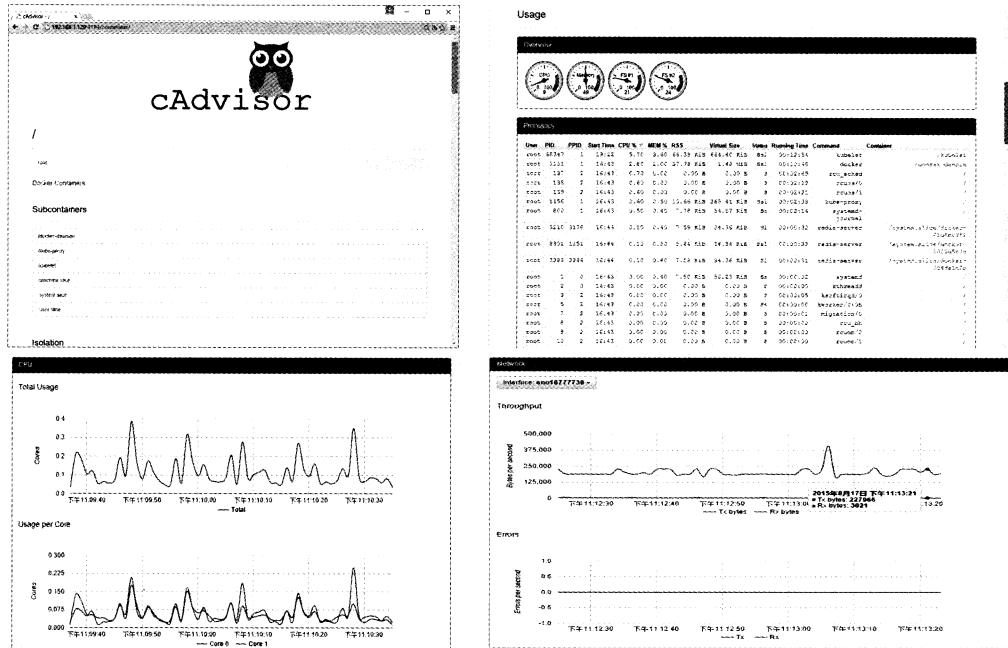


图5.7 主机的性能监控页面

通过Docker Containers链接可以查看容器列表及每个容器的性能数据，如图5.8所示。

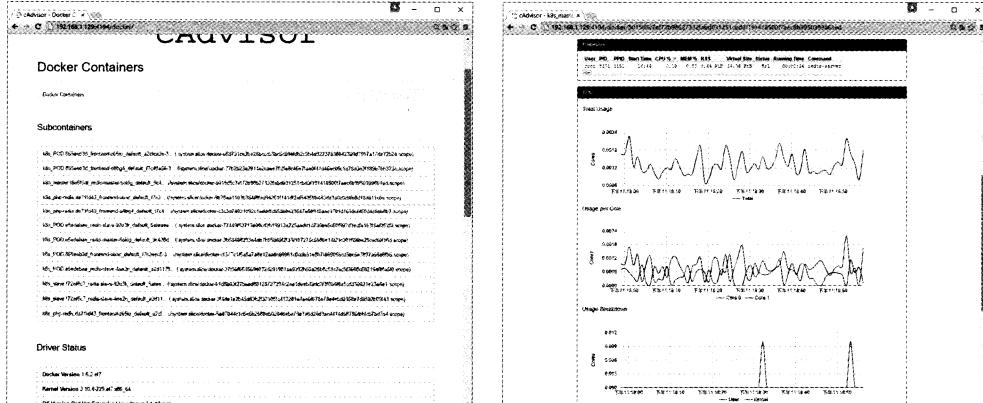


图5.8 容器的性能监控页面

此外，cAdvisor也提供了REST API供客户端远程调用，主要是为了定制开发，API返回的数据格式为JSON，可以采用如下URL来访问：

```
http://<hostname>:<port>/api/<version>/<request>
```

例如，通过URL `http://192.168.18.3:4194/api/v1.3/machine` 可以获取主机的相关信息：

```
{
  "num_cores":2,
  "cpu_frequency_khz":2793544,
  "memory_capacity":1915408384,
  "machine_id":"0f6233d8256a4ec1a673640e04b8344a",
  "system_uuid":"564D188F-8E82-21C0-6E89-176E2C51EBB5",
  "boot_id":"a03d00d8-ca9c-4d74-a674-ebf5dfbc69d9",
  "filesystems":[
    {
      "device":"/dev/mapper/rhel-root",
      "capacity":18746441728
    },
    {
      "device":"/dev/sdal",
      "capacity":520794112
    }
  ],
  "disk_map":{
    "253:0":{
      "name":"dm-0",
      "major":253,
      "minor":0,
      "size":2147483648,
      "scheduler":"none"
    },
    .....
  },
  "network_devices":[
    {
      "name":"eno16777736",
      "mac_address":"00:0c:29:51:eb:b5",
      "speed":1000,
      "mtu":1500
    }
  ],
  "topology":[
    {
      "node_id":0,
      "memory":2146947072,
      "cores":[
        {
          "id":0,
          "socket_id":0,
          "core_id":0,
          "physical_id":0,
          "thread_id":0
        }
      ]
    }
  ]
}
```

```
        "core_id":0,
        "thread_ids":[
            0
        ],
        "caches":null
    },
    .....
],
"caches": [
    {
        "size":6291456,
        "type":"Unified",
        "level":3
    }
]
}
]
```

通过下面的 URL 则可以获取节点上最新（1 分钟内）的容器的性能数据：<http://192.168.1.129:4194/api/v1.3/subcontainers/system.slice/docker-5015d5c7ef72b98627332fabd031251cbd3f191418500f7aec6b9950399661ed.scope>。

结果为：

```
[
{
    "name":"/system.slice/docker-5015d5c7ef72b98627332fabd031251cbd3f191418500f7aec6b9950399661ed.scope",
    "aliases":[
        "k8s_master.f8a6f6df_redis-master-6okig_default_9c428d4f-4167-11e5-afe7-000c2921ba71_5dce2f85",
        "5015d5c7ef72b98627332fabd031251cbd3f191418500f7aec6b9950399661ed"
    ],
    "namespace":"docker",
    "spec":{
        "creation_time":"2015-08-17T08:44:27.401122502Z",
        "labels":{
            "io.kubernetes.pod.name":"default/Redis-master-6okig"
        },
        "has_cpu":true,
        "cpu":{
            "limit":2,
            "max_limit":0,
            "mask":"0-1"
        },
        "has_memory":true,
```

```
"memory":{  
    "limit":18446744073709552000,  
    "swap_limit":18446744073709552000  
},  
"has_network":true,  
"has_filesystem":false,  
"has_diskio":true  
},  
"stats": [  
    {  
        "timestamp": "2015-08-18T00:54:26.167988505+08:00",  
        "cpu": {  
            "usage": {  
                "total": 43121463207,  
                "per_cpu_usage": [  
                    21578091763,  
                    21543371444  
                ],  
                "user": 410000000,  
                "system": 13620000000  
            },  
            "load_average": 0  
        },  
        "diskio": {  
            "io_service_bytes": [  
                {  
                    "major": 253, "minor": 14,  
                    "stats": {  
                        "Async": 8036352, "Read": 8036352, "Sync": 0, "Total": 8036352, "Write": 0  
                    }  
                }  
            ]  
        },  
        "io_serviced": [  
            {  
                "major": 8,  
                "minor": 0,  
                "stats": {  
                    "Async": 0,  
                    ....  
                }  
            ]  
        },  
        "memory": {  
            "usage": 16748544,  
            "working_set": 9297920,  
            "container_data": {  
                "pgfault": 882,  
                "pgmajfault": 8  
            }  
        }  
    }  
]
```

```

        },
        "hierarchical_data": {
            "pgfault": 882,
            "pgmajfault": 8
        }
    },
    "network": {
        "name": ""
    },
    "rx_bytes": 0, "rx_packets": 0, "rx_errors": 0, "rx_dropped": 0, "tx_bytes": 0, "tx_packets": 0, "tx_errors": 0, "tx_dropped": 0
},
"task_stats": {
    "nr_sleeping": 0, "nr_running": 0, "nr_stopped": 0, "nr_uninterruptible": 0, "nr_io_wait": 0
}
}
.....
]
}
]

```

容器的性能数据对于集群监控非常有用，系统管理员可以根据 cAdvisor 提供的数据进行分析和告警。不过，由于 cAdvisor 是在每台 Node 上运行的，只能采集本机的性能指标数据，所以系统管理员需要对每台 Node 主机单独监控。

针对大型集群，Kubernetes 建议使用几个开源软件组成的集成解决方案来实现对整个集群的监控。这些开源软件包括 Heapster、InfluxDB 及 Grafana 等。

2. Heapster+Influxdb+Grafana 集群性能监控平台搭建

根据前面的说明，cAdvisor 集成在 kubelet 中，运行在每个 Node 上，所以一个 cAdvisor 仅能对一台 Node 进行监控。在大规模容器集群中，需要对所有 Node 和全部容器进行性能监控，Kubernetes 建议使用一套工具来实现集群性能数据的采集、存储和展示：Heapster、InfluxDB 和 Grafana。

- ◎ **Heapster：**对集群中各 Node 上 cAdvisor 的数据采集汇聚的系统，通过访问每个 Node 上 kubelet 的 API，再通过 kubelet 调用 cAdvisor 的 API 来采集该节点上所有容器的性能数据。Heapster 对性能数据进行聚合，并将结果保存到后端存储系统中。Heaspter 支持多种后端存储系统，包括 memory（保存在内存中）、InfluxDB、BigQuery、谷歌云平台提供的 Google Cloud Monitoring (<https://cloud.google.com/monitoring/>) 和 Google Cloud Logging (<https://cloud.google.com/logging/>) 等。Heapster 项目的主页为 <https://github.com/kubernetes/heapster>。

- ◎ **InfluxDB**: 是分布式时序数据库（每条记录都带有时间戳属性），主要用于实时数据采集、事件跟踪记录、存储时间图表、原始数据等。InfluxDB 提供了 REST API 用于数据的存储和查询。InfluxDB 的主页为 <http://influxdb.com>。
- ◎ **Grafana**: 通过 Dashboard 将 InfluxDB 中的时序数据展现成图表或曲线等形式，便于运维人员查看集群的运行状态。Grafana 的主页为 <http://grafana.org>。

基于 heapster+influxdb+grafana 的集群监控系统总体架构如图 5.9 所示。

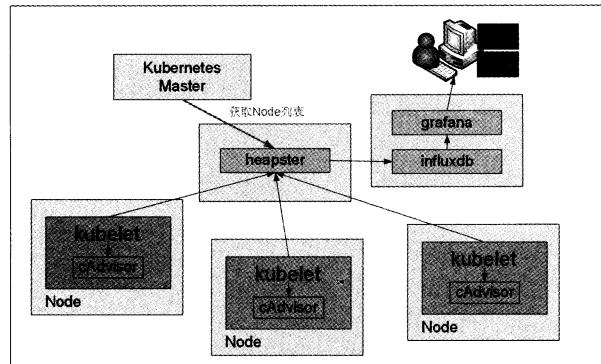


图 5.9 Heapster 集群监控系统架构图

Heapster、InfluxDB 和 Grafana 均以 Pod 的形式启动和运行。由于 Heapster 需要与 Kubernetes Master 进行安全连接，所以需要设置 Master 的 CA 证书安全策略（参见第 2 章的说明）。

1) 部署 Heapster、InfluxDB、Grafana 容器应用

先创建它们的 Service:

heapster-service.yaml

```

apiVersion: v1
kind: Service
metadata:
  labels:
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: Heapster
  name: heapster
  namespace: kube-system
spec:
  ports:
    - port: 80
      targetPort: 8082
  selector:
    k8s-app: heapster

```

influxdb-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels: null
  name: monitoring-InfluxDB
  namespace: kube-system
spec:
  type: NodePort
  ports:
    - name: http
      port: 8083
      targetPort: 8083
      nodePort: 8083
    - name: api
      port: 8086
      targetPort: 8086
      nodePort: 8086
  selector:
    name: influxGrafana
```

注意，这里使用 `type=NodePort` 将 InfluxDB 暴露在宿主机 Node 的端口上，以便我们使用浏览器对其进行访问。

grafana-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  labels:
    kubernetes.io/name: monitoring-Grafana
    kubernetes.io/cluster-service: "true"
  name: monitoring-Grafana
  namespace: kube-system
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 8085
  selector:
    name: influxGrafana
```

同样，使用 `type=NodePort` 将 Grafana 暴露在 Node 的端口上，以便客户端的浏览器对其进行访问。

使用 `kubectl create` 命令创建 Services：

```
$ kubectl create -f heapster-service.yaml  
$ kubectl create -f InfluxDB-service.yaml  
$ kubectl create -f Grafana-service.yaml
```

在创建 heapster 容器之前，先创建 InfluxDB 和 Grafana 的 RC，这两个容器将运行在同一个 Pod 中：

```
influxdb-grafana-controller-v3.yaml  
  
apiVersion: v1  
kind: ReplicationController  
metadata:  
  name: monitoring-influxdb-grafana-v3  
  namespace: kube-system  
  labels:  
    k8s-app: influxGrafana  
    version: v3  
    kubernetes.io/cluster-service: "true"  
spec:  
  replicas: 1  
  selector:  
    k8s-app: influxGrafana  
    version: v3  
  template:  
    metadata:  
      labels:  
        k8s-app: influxGrafana  
        version: v3  
        kubernetes.io/cluster-service: "true"  
    spec:  
      containers:  
        - image: gcr.io/google_containers/heapster_influxdb:v0.5  
          name: influxdb  
          resources:  
            # keep request = limit to keep this container in guaranteed class  
            limits:  
              cpu: 100m  
              memory: 500Mi  
            requests:  
              cpu: 100m  
              memory: 500Mi  
          ports:  
            - containerPort: 8083  
            - containerPort: 8086  
          volumeMounts:  
            - name: influxdb-persistent-storage  
              mountPath: /data
```

```
- image: gcr.io/google_containers/heapster_grafana:v2.6.0-2
  name: grafana
  resources:
    limits:
      cpu: 100m
      memory: 100Mi
    requests:
      cpu: 100m
      memory: 100Mi
  env:
    # This variable is required to setup templates in Grafana.
    - name: INFLUXDB_SERVICE_URL
      value: http://monitoring-influxdb:8086
    # The following env variables are required to make Grafana accessible
via
recommend
the grafana
# the kubernetes api-server proxy. On production clusters, we
# removing these env variables, setup auth for grafana, and expose
# service using a LoadBalancer or a public IP.
- name: GF_AUTH_BASIC_ENABLED
  value: "false"
- name: GF_AUTH_ANONYMOUS_ENABLED
  value: "true"
- name: GF_AUTH_ANONYMOUS_ORG_ROLE
  value: Admin
- name: GF_SERVER_ROOT_URL
  value:
/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/
  volumeMounts:
    - name: grafana-persistent-storage
      mountPath: /var
  volumes:
    - name: influxdb-persistent-storage
      emptyDir: {}
    - name: grafana-persistent-storage
      emptyDir: {}
```

注意，Grafana 容器环境变量 INFLUXDB_SERVICE_URL 设置为 InfluxDB 服务的所在地址。由于 Grafana 与 InfluxDB 处于同一个 Pod 中，所以 Grafana 使用 127.0.0.1 或 localhost 也可以访问到 InfluxDB 服务。

使用 kubectl create 命令创建该 RC：

```
$ kubectl create -f influxdb-grafana-controller-v3.yaml
```

通过 kubectl get pods --namespace=kube-system 确认 Pod 成功启动：

```
# kubectl get pods --namespace=kube-system
NAME                               READY   STATUS    RESTARTS   AGE
monitoring-influxdb-grafana-v3-uu730   2/2     Running   0          4m
```

创建 heapster 容器，v1.1.0 版本的 heapster 由 4 个容器组合为一个 Pod:

heapster-controller-v1.1.0.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: heapster-v1.1.0
  namespace: kube-system
  labels:
    k8s-app: heapster
    kubernetes.io/cluster-service: "true"
    version: v1.1.0
spec:
  replicas: 1
  selector:
    matchLabels:
      k8s-app: heapster
      version: v1.1.0
  template:
    metadata:
      labels:
        k8s-app: heapster
        version: v1.1.0
    spec:
      # 4 containers, 2 heapsters, 2 resizer
      containers:
        - image: gcr.io/google_containers/heapster:v1.1.0
          name: heapster
          resources:
            # keep request = limit to keep this container in guaranteed class
            limits:
              cpu: 100m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          command:
            - /heapster
            - --source=kubernetes.summary_api:'192.168.18.3:8080'
            - --sink=influxdb:http://monitoring-influxdb:8086
            - --metric_resolution=60s
        - image: gcr.io/google_containers/heapster:v1.1.0
          name: eventer
```

```
resources:
  # keep request = limit to keep this container in guaranteed class
  limits:
    cpu: 100m
    memory: 200Mi
  requests:
    cpu: 100m
    memory: 200Mi
  command:
    - /eventer
    - --source=kubernetes:'192.168.18.3:8080'
    - --sink=influxdb:http://monitoring-influxdb:8086
- image: gcr.io/google_containers/addon-resizer:1.3
  name: heapster-nanny
resources:
  limits:
    cpu: 50m
    memory: 100Mi
  requests:
    cpu: 50m
    memory: 100Mi
env:
  - name: MY_POD_NAME
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: MY_POD_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
command:
  - /pod_nanny
  - --cpu=100m
  - --extra-cpu=0m
  - --memory=200Mi
  - --extra-memory=4Mi
  - --threshold=5
  - --deployment=heapster-v1.1.0
  - --container=heapster
  - --poll-period=300000
  - --estimator=exponential
- image: gcr.io/google_containers/addon-resizer:1.3
  name: eventer-nanny
resources:
  limits:
    cpu: 50m
    memory: 100Mi
```

```

requests:
  cpu: 50m
  memory: 100Mi
env:
- name: MY_POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
- name: MY_POD_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
command:
- /pod_nanny
- --cpu=100m
- --extra-cpu=0m
- --memory=200Mi
- --extra-memory=500Ki
- --threshold=5
- --deployment=heapster-v1.1.0
- --container=eventer
- --poll-period=300000
- --estimator=exponential

```

Heapster 需要设置的启动参数如下。

(1) -source

配置采集来源，为 Master URL 地址：

```
--source=kubernetes.summary_api:'192.168.18.3:8080'
```

(2) -sink

配置后端存储系统，使用 InfluxDB 系统：

```
--sink=InfluxDB:http://monitoring-InfluxDB:8086
```

(3) --metric_resolution

性能指标的精度，60s 表示将过去 60 秒的数据进行汇聚再进行存储。

其他参数可以通过进入 heapster 容器执行 # heapster -help 命令查看和设置。

注意，URL 中的主机名地址使用的是 InfluxDB 的 Service 名字，这需要 DNS 服务正常工作，如果没有配置 DNS 服务，则也可以使用 Service 的 ClusterIP 地址。

值得说明的是，InfluxDB 服务的名称没有加上命名空间，是因为 Heapster 服务与 InfluxDB 服务属于相同的命名空间 kube-system。当然，使用带上命名空间的全服务名也是可以的，例如 <http://monitoring-influxdb.kube-system:8086>。

使用 `kubectl create` 命令完成创建该 RC：

```
$ kubectl create -f heapster-controller-v1.1.0.yaml
```

通过 `kubectl get pods --namespace=kube-system` 确认 Pod 成功启动：

```
# kubectl get deployment --namespace=kube-system
NAME          READY   STATUS    RESTARTS   AGE
heapster-v1.1.0-1895667918-guisl   4/4     Running   0          3m
```

查看 heapster 的日志，确保 heapster 成功在 influxdb 数据库中创建名为 k8s 的数据库：

```
# kubectl logs heapster-v1.1.0-1895667918-guisl -c heapster
--namespace=kube-system
I0706 09:36:15.313587      1 heapster.go:65] /heapster
--source=kubernetes.summary_api:'192.168.18.3:8080'
--sink=influxdb:http://monitoring-influxdb:8086 --metric_resolution=60s
I0706 09:36:15.313849      1 heapster.go:66] Heapster version 1.1.0
I0706 09:36:15.314347      1 configs.go:60] Using Kubernetes client with master
"https://169.169.0.1:443" and version "v1"
I0706 09:36:15.314371      1 configs.go:61] Using kubelet port 10255
I0706 09:36:15.512107      1 influxdb.go:223] created influxdb sink with options:
host:monitoring-influxdb:8086 user:root db:k8s
I0706 09:36:15.512154      1 heapster.go:92] Starting with InfluxDB Sink
I0706 09:36:15.512163      1 heapster.go:92] Starting with Metric Sink
I0706 09:36:16.414060      1 heapster.go:171] Starting heapster on port 8082
```

2) 查询 InfluxDB 数据库中的数据

让我们先通过 InfluxDB 的管理页面查看数据。

由于设置 InfluxDB 服务会暴露到物理 Node 节点上，所以我们可以通过任一 Node 的 8083 端口访问 InfluxDB 数据库提供的管理页面，如图 5.10 所示。通过右上角齿轮按钮可以修改连接属性（用于 influxdb service 设置为非默认端口号的时候）。单击右上角的 Database 下拉列表可以选择数据库，heapster 创建的数据库名为 k8s。

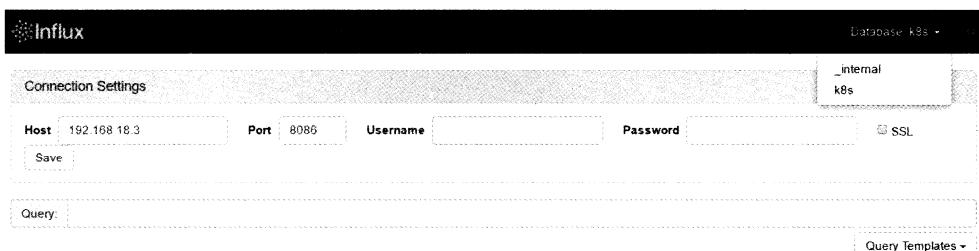


图 5.10 InfluxDB 管理页面

在 Query 输入框中输入“SHOW MEASUREMENTS”，即可查看所有的 measurements（序列表）。图 5.11 显示了部分 measurements。

The screenshot shows the InfluxDB interface with the title 'measurements'. Below it is a table with two columns: 'name'. The listed items are:

- cpu/limit
- cpu/node_reservation
- cpu/node_utilization
- cpu/request
- cpu/usage
- cpu/usage_rate
- filesystem/available
- filesystem/limit
- filesystem/usage

图 5.11 show measurements 结果页面

heapster 采集的全部 metric（性能指标）如表 5.6 所示。

表 5.6 heapster 采集的 metric

metric 名称	说 明
cpu/limit	CPU hard limit, 单位为毫秒
cpu/node_reservation	Node 保留的 CPU Share
cpu/node_utilization	Node 的 CPU 使用时间
cpu/request	CPU request, 单位为毫秒
cpu/usage	全部 Core 的 CPU 累计使用时间
cpu/usage_rate	全部 Core 的 CPU 累计使用率, 单位为毫秒
filesystem/usage	文件系统已用的空间, 单位为字节
filesystem/limit	文件系统总空间限制, 单位为字节
filesystem/available	文件系统可用的空间, 单位为字节
memory/limit	Memory hard limit, 单位为字节
memory/major_page_faults	major page faults 数量
memory/major_page_faults_rate	每秒的 major page faults 数量
memory/node_reservation	Node 保留的内存 Share
memory/node_utilization	Node 的内存使用值
memory/page_faults	page faults 数量
memory/page_faults_rate	每秒的 page faults 数量

续表

metric 名称	说 明
memory/request	Memory request，单位为字节
memory/usage	总内存使用量
memory/working_set	总的 Working set usage，Working set 是指不会被 kernel 移除的内存
network/rx	累计接收的网络流量字节数
network/rx_errors	累计接收的网络流量错误数
network/rx_errors_rate	每秒接收的网络流量错误数
network/rx_rate	每秒接收的网络流量字节数
network/tx	累计发送的网络流量字节数
network/tx_errors	累计发送的网络流量错误数
network/tx_errors_rate	每秒发送的网络流量错误数
network/tx_rate	每秒发送的网络流量字节数
uptime	容器启动总时长

每个 metric 可以看作一张数据库表，表中每条记录由一组 label 组成，可以看作字段，如表 5.7 所示。

表 5.7 metric 的各 label

Label 名称	说 明
pod_id	系统生成的 Pod 唯一名称
pod_name	用户指定的 Pod 名称
pod_namespace	Pod 所属的 namespace
container_base_image	容器的镜像名称
container_name	用户指定的容器名称
host_id	用户指定的 Node 主机名
hostname	容器运行所在主机名
labels	逗号分隔的 Label 列表
namespace_id	Pod 所属的 namespace 的 UID
resource_id	资源 ID

可以使用标准 SQL SELECT 语句对每个 metric 进行查询，例如查询 CPU 的使用时间：

```
select * from "cpu/usage" limit 10
```

结果如图 5.12 所示。

cpu/usage													
time	container_base_image	container_name	host_id	hostname	label	namespace_id	namespacename	nodeid	pod_id	pod_name	pod_namespace	type	value
2016-08-06T21:32:00Z	"gcr.io/google_containers/heptcd:v1.0"	"heptcd"	"10s-node-1"	"10s-node-1"	"10s-app-heptcd:pod-template-hash:1895667918,version:v1.0"	"795be852"	"kube-system"	"10s-node-1"	"31e1f644-5c10-11e6-ba0c-000c29d2102"	"heptcd"	"kube-system"	"pod_container"	4204057
2016-08-06T21:32:00Z	"gcr.io/google_containers/etcd:2.3.6-0.20.1"	"etcd0"	"10s-node-1"	"10s-node-1"	"10s-app-etcd:ds.kubeletes.kubernetes-service=true,version:v1.1"	"459011e6-ba0c-000c29d2102"	"kube-system"	"10s-node-1"	"ca768d13-5c10-11e6-ba0c-000c29d2102"	"etcd0"	"kube-dns"	"pod_system"	234138702151
2016-08-06T21:32:00Z	"gcr.io/google_containers/etcd:2.3.6-0.20.1"	"etcd1"	"10s-node-1"	"10s-node-1"	"10s-app-etcd:ds.kubeletes.kubernetes-service=true,version:v1.1"	"795be852"	"kube-system"	"10s-node-1"	"31e1f644-5c10-11e6-ba0c-000c29d2102"	"etcd1"	"kube-dns"	"pod_system"	26326516079
2016-08-06T21:32:00Z	"gcr.io/google_containers/heapster:v1.3"	"heapster"	"10s-node-1"	"10s-node-1"	"10s-app-heapster:pod-template-hash:1895667918,version:v1.0"	"795be852"	"kube-system"	"10s-node-1"	"31e1f644-5c10-11e6-ba0c-000c29d2102"	"heapster"	"kube-system"	"pod_container"	37616862
2016-08-06T21:32:00Z	"gcr.io/google_containers/addon-resizer:1.3"	"addon-resizer"	"10s-node-1"	"10s-node-1"	"10s-app-addon-resizer:pod-template-hash:1895667918,version:v1.0"	"795be852"	"kube-system"	"10s-node-1"	"31e1f644-5c10-11e6-ba0c-000c29d2102"	"addon-resizer"	"kube-system"	"pod_container"	48032543619721
2016-08-06T21:32:00Z	"gcr.io/google_containers/heptcd:v1.0"	"heptcd"	"10s-node-1"	"10s-node-1"	"10s-app-heptcd:pod-template-hash:1895667918,version:v1.0"	"795be852"	"kube-system"	"10s-node-1"	"31e1f644-5c10-11e6-ba0c-000c29d2102"	"heptcd"	"kube-system"	"pod_container"	41665260
2016-08-06T21:32:00Z	"gcr.io/google_containers/kubelet:v1.6.0-0.20.1"	"kubelet0"	"10s-node-1"	"10s-node-1"	"10s-app-kubelet:ds.kubeletes.kubernetes-service=true,version:v1.1"	"459011e6-ba0c-000c29d2102"	"kube-system"	"10s-node-1"	"ca768d13-5c10-11e6-ba0c-000c29d2102"	"kubelet0"	"kube-dns"	"pod_system"	17674955092
2016-08-06T21:32:00Z	"gcr.io/google_containers/kubelet:v1.6.0-0.20.1"	"kubelet1"	"10s-node-1"	"10s-node-1"	"10s-app-kubelet:ds.kubeletes.kubernetes-service=true,version:v1.1"	"795be852"	"kube-system"	"10s-node-1"	"31e1f644-5c10-11e6-ba0c-000c29d2102"	"kubelet1"	"kube-dns"	"pod_system"	41595166
2016-08-06T21:32:00Z	"gcr.io/google_containers/infinitd:0.5"	"infinitd"	"10s-node-1"	"10s-node-1"	"10s-app-infinitd:ds.infinitd.infra.kubernetes.io/cluster-service=true,version:v3"	"795be852"	"kube-system"	"10s-node-1"	"31e1f644-5c10-11e6-ba0c-000c29d2102"	"infinitd"	"kube-system"	"pod_container"	427297277

图 5.12 查询 cpu/usage 结果页面

3) Grafana 页面查看和操作

访问 Grafana 服务需要通过 Master 代理模式进行访问, URL 地址为 <http://192.168.18.3:8080/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana/>。

在 grafana 主页可以查看监控数据的图表展示画面。如图 5.13 所示为 Cluster 集群的整体信息, 以折线图的形式展示了集群范围内各 Node 的 CPU 使用率、内存使用情况等信息。

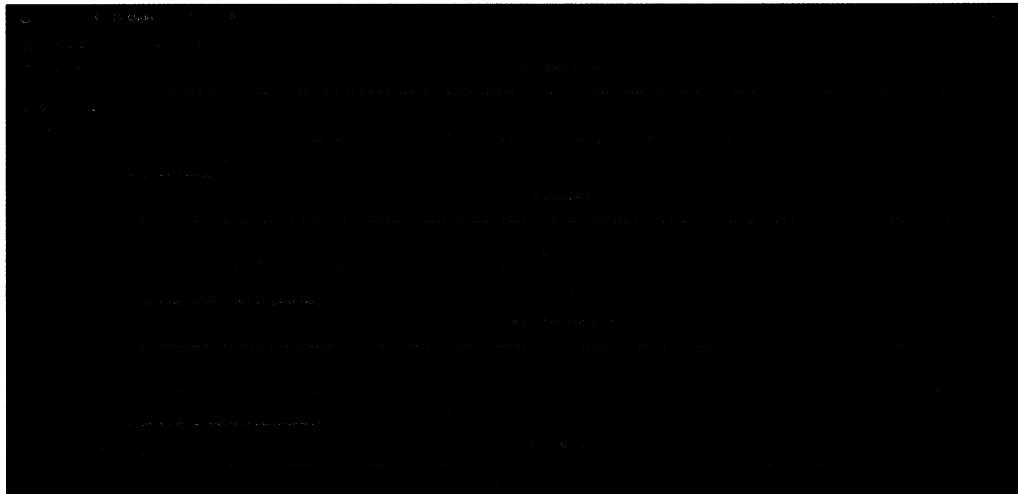


图 5.13 Grafana Cluster 监控页面

图 5.14 显示的是所有 Pod 的信息，以折线图的形式展示了集群范围内各 Pod 的 CPU 使用率、内存使用情况、网络流量、文件系统使用情况等信息。

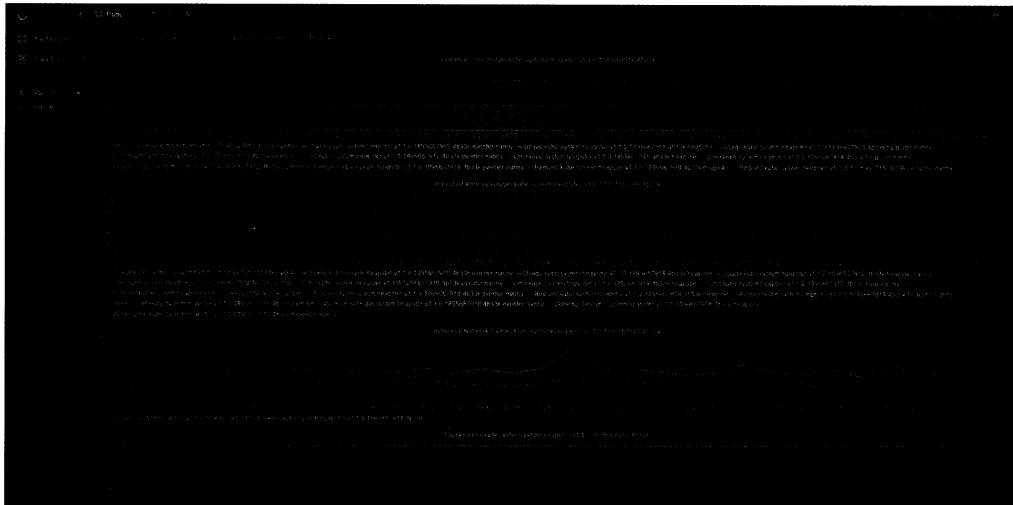


图 5.14 Grafana Pod 监控页面

Grafana 页面上的每个图表都可以进行编辑，在标题上单击鼠标，点击“Edit”进入编辑页面，可以对每个 metric 进行个性化设置，例如查询的表名、字段名、汇总计算等，如图 5.15 所示。



图 5.15 编辑折线图

到此，基于 heapster+influxdb+grafana 的 Kubernetes 集群监控系统就搭建完成了。

5.1.7 kubelet 的垃圾回收 (GC) 机制

Kubernetes 集群中的垃圾回收 (Garbage Collection, 简称 GC) 机制由 kubelet 完成。kubelet 定期清理不再使用的容器和镜像，每分钟进行一次容器 GC 操作，每 5 分钟进行一次镜像 GC 操作。

1. 容器 (Container) 的 GC 设置

能够被 GC 清理的容器只能是仅由 kubelet 管理的容器。在 kubelet 所在的 Node 上直接通过 docker run 创建的容器将不会被 kubelet 进行 GC 清理操作。

kubelet 的以下 3 个启动参数用于设置容器 GC 的条件。

- ◎ **--minimum-container-ttl-duration:** 已停止的容器在被清理之前的最小存活时间，例如“300ms”“10s”或“2h45m”，超过此存活时间的容器将被标记为可被 GC 清理，默认值为 1 分钟。
- ◎ **--maximum-dead-containers-per-container:** 以 Pod 为单位的可以保留的已停止的（属于同一 Pod 的）容器集的最大数量。有时，Pod 中容器运行失败或者健康检查失败后，会被 kubelet 自动重启，这将产生一些停止的容器。默认值为 2。
- ◎ **--maximum-dead-containers:** 在本 Node 上保留的已停止容器的最大数量，由于停止的容器也会消耗磁盘空间，所以超过该上限以后，kubelet 会自动清理已停止的容器以释放磁盘空间，默认值为 240。

如果需要关闭针对容器的 GC 操作，则可以将--minimum-container-ttl-duration 设置为 0，将--maximum-dead-containers-per-container 和--maximum-dead-containers 设置为负数。

2. 镜像 (Image) 的 GC 设置

Kubernetes 系统中通过 imageController 和 kubelet 中集成的 cAdvisor 共同管理镜像的生命周期，主要根据本 Node 的磁盘使用率来触发镜像的 GC 操作。

kubelet 的以下 3 个启动参数用于设置镜像 GC 的条件。

- ◎ **--minimum-image-ttl-duration:** 不再使用的镜像在被清理之前的最小存活时间，例如“300ms”“10s”或“2h45m”，超过此存活时间的镜像被标记为可被 GC 清理，默认值为两分钟。
- ◎ **--image-gc-high-threshold:** 当磁盘使用率达到该值时，触发镜像的 GC 操作，默认值为 90%。
- ◎ **--image-gc-low-threshold:** 当磁盘使用率降到该值时，GC 操作结束，默认值为 80%。

删除镜像的机制为：当磁盘使用率达到 `image-gc-high-threshold`（例如 90%）时触发，GC 操作从最久未使用（Least Recently Used）的镜像开始删除，直到磁盘使用率降为 `image-gc-low-threshold`（例如 80%）或没有镜像可删为止。

5.2 Kubernetes 高级案例

本节将对 ElasticSearch 日志管理平台的部署、Cassandra 集群的部署及 Kubernetes 中容器的高级应用进行说明。

5.2.1 ElasticSearch 日志搜集查询和展现案例

在 Kubernetes 集群环境中，一个完整的应用或服务都会涉及为数众多的组件运行，各组件所在的 Node 及实例数量都是可变的。日志子系统如果不做集中化管理，则会给系统的运维支撑造成很大的困难，因此有必要在集群层面对日志进行统一的收集和检索等工作。

容器中输出到控制台的日志，都会以`*-json.log` 的命名方式保存在`/var/lib/docker/containers/`目录之下，这样就给了我们进行日志采集和后续处理的基础。

Kubernetes 推荐采用 Fluentd+ElasticSearch+Kibana 完成对日志的采集、查询和展现工作。

在部署系统之前，需要以下两个前提条件。

- ◎ API Server 正确配置了 CA 证书。
- ◎ DNS 服务启动运行。

1. 系统部署架构

系统的逻辑架构如图 5.16 所示。

在各 Node 上运行一个 Fluentd 容器，对本节点`/var/log` 和`/var/lib/docker/containers` 两个目录下的日志进程采集，然后汇总到 ElasticSearch 集群，最终通过 Kibana 完成和用户的交互工作。

这里有一个特殊的需求，Fluentd 必须在每个 Node 上运行一份，为了满足这一需要，我们有以下几种不同的方式来部署 Fluentd。

- ◎ 直接在 Node 主机上部署 Fluentd。
- ◎ 利用 kubelet 的`--config` 参数，为每个 Node 加载 Fluentd Pod。
- ◎ 利用 DaemonSet 来让 Fluentd Pod 在每个 Node 上运行。

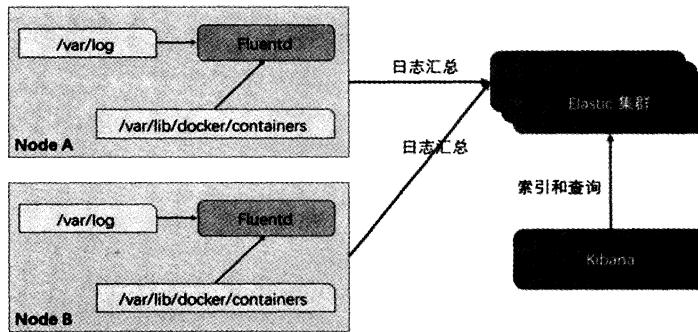


图 5.16 Fluentd+ElasticSearch+Kibana 系统逻辑架构图

目前官方推荐的包括Fluentd、Logstash等日志或者监控类的Pod的运行方式就是DaemonSet方式，因此本节我们也以这一方式进行配置。

2. 创建ElasticSearch RC 和 Service

ElasticSearch 的 RC 和 Service 定义：

```
elasticsearch-rc-svc.yml
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: elasticsearch-logging-v1
  namespace: kube-system
  labels:
    k8s-app: elasticsearch-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 2
  selector:
    k8s-app: elasticsearch-logging
    version: v1
  template:
    metadata:
      labels:
        k8s-app: elasticsearch-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
        - image: gcr.io/google_containers/elasticsearch:1.8
          name: elasticsearch-logging
```

```
resources:
  # keep request = limit to keep this container in guaranteed class
  limits:
    cpu: 100m
  requests:
    cpu: 100m
ports:
- containerPort: 9200
  name: db
  protocol: TCP
- containerPort: 9300
  name: transport
  protocol: TCP
volumeMounts:
- name: es-persistent-storage
  mountPath: /data
volumes:
- name: es-persistent-storage
  emptyDir: {}

---
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch-logging
  namespace: kube-system
  labels:
    k8s-app: elasticsearch-logging
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Elasticsearch"
spec:
  ports:
  - port: 9200
    protocol: TCP
    targetPort: db
  selector:
    k8s-app: elasticsearch-logging
```

执行 `kubectl create -f elastic-search.yml` 命令完成创建。

命令成功执行后，首先验证 Pod 的运行情况。通过 `kubectl get pods --namespace=kube-system` 获取运行中的 Pod：

```
# kubectl get pods --namespace=kube-system
NAMESPACE      NAME                               READY   STATUS    RESTARTS   AGE
kube-system    elasticsearch-logging-v1-59qvp     1/1    Running   0          18h
kube-system    elasticsearch-logging-v1-xnv14     1/1    Running   0          18h
```

接下来通过 ElasticSearch 的页面验证其功能。

执行# kubectl cluster-info 命令获取 ElasticSearch 服务的地址:

```
# kubectl cluster-info
Elasticsearch is running at
http://192.168.18.3:8080/api/v1/proxy/namespaces/kube-system/services/elasticsearch-logging
```

接下来使用 # kubectl proxy 命令对 apiserver 进行代理，成功执行后输出如下：

```
# kubectl proxy
Starting to serve on 127.0.0.1:8001
```

这样我们就可以在浏览器上访问 URL 地址 http://192.168.18.3:8001/api/v1/proxy/namespaces/kube-system/services/elasticsearch-logging，来验证 ElasticSearch 的运行情况了，返回的内容是一个 JSON 文档：

```
{
  "status": 200,
  "name": "Emplate",
  "cluster_name": "kubernetes-logging",
  "version": {
    "number": "1.5.2",
    "build_hash": "62ff9868b4c8a0c45860bebb259e21980778ab1c",
    "build_timestamp": "2015-04-27T09:21:06Z",
    "build_snapshot": false,
    "lucene_version": "4.10.4"
  },
  "tagline": "You Know, for Search"
}
```

3. 在每个 Node 上启动 Fluentd

Fluentd 的 DaemonSet 定义如下：

```
fluentd-ds.yaml
---
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: fluentd-cloud-logging
  namespace: kube-system
  labels:
    k8s-app: fluentd-cloud-logging
spec:
  template:
    metadata:
      namespace: kube-system
      labels:
```

```
k8s-app: fluentd-cloud-logging
spec:
  containers:
    - name: fluentd-cloud-logging
      image: gcr.io/google_containers/fluentd-elasticsearch:1.17
      resources:
        limits:
          cpu: 100m
          memory: 200Mi
      env:
        - name: FLUENTD_ARGS
          value: -q
      volumeMounts:
        - name: varlog
          mountPath: /var/log
          readOnly: false
        - name: containers
          mountPath: /var/lib/docker/containers
          readOnly: false
      volumes:
        - name: containers
          hostPath:
            path: /var/lib/docker/containers
        - name: varlog
          hostPath:
            path: /var/log
```

通过 kubectl create 命令创建 Fluentd 容器：

```
# kubectl create -f fluentd-ds.yaml
```

查看创建的结果：

```
# kubectl get daemonset
NAME           DESIRED   CURRENT   NODE-SELECTOR   AGE
fluentd-cloud-logging   3         3         <none>       1h

# kubectl get pods
NAMESPACE     NAME           READY   STATUS    RESTARTS   AGE
fluentd-cloud-logging-7tw9z   1/1     Running   0          18h
fluentd-cloud-logging-aqdn1   1/1     Running   0          18h
fluentd-cloud-logging-o4usx   1/1     Running   0          18h
```

结果显示 Fluentd DaemonSet 正常运行，启动 3 个 Pod，与集群中的 Node 数量一致。

接下来，使用 # kubectl logs fluentd-cloud-logging-7tw9z 命令查看 Pod 的日志，在 ElasticSearch 正常工作的情况下，我们会看到类似下面这样的日志内容：

```
# kubectl logs fluentd-cloud-logging-7tw9z
Connection opened to Elasticsearch cluster =>
```

```
{:host=>"elasticsearch-logging", :port=>9200, :scheme=>"http"}
```

说明 Fluentd 与 ElasticSearch 已经正确建立了连接。

4. 运行 Kibana

到此我们已经运行了 ElasticSearch 和 Fluentd，数据的采集和汇聚过程已经完成，接下来就是使用 Kibana 来展示和操作数据了。

Kibana 的 RC 和 Service 定义如下：

```
kibana-rc-svc.yml
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: kibana-logging-v1
  namespace: kube-system
  labels:
    k8s-app: kibana-logging
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kibana-logging
    version: v1
  template:
    metadata:
      labels:
        k8s-app: kibana-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
        - name: kibana-logging
          image: gcr.io/google_containers/kibana:1.3
          resources:
            # keep request = limit to keep this container in guaranteed class
            limits:
              cpu: 100m
            requests:
              cpu: 100m
      env:
        - name: "ELASTICSEARCH_URL"
          value: "http://elasticsearch-logging:9200"
      ports:
```

```
- containerPort: 5601
  name: ui
  protocol: TCP
---
apiVersion: v1
kind: Service
metadata:
  name: kibana-logging
  namespace: kube-system
  labels:
    k8s-app: kibana-logging
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "Kibana"
spec:
  ports:
  - port: 5601
    protocol: TCP
    targetPort: ui
  selector:
    k8s-app: kibana-logging
```

通过 `kubectl create -f kibana-rc-svc.yml` 命令创建 Kibana 的 RC 和 Service:

```
# kubectl create -f kibana-rc-svc.yml
replicationcontroller "kibana-logging-v1" created
service "kibana-logging" created
```

查看 Kibana 的运行情况:

```
# kubectl get pods
NAMESPACE     NAME           READY   STATUS    RESTARTS   AGE
default       kibana-logging-v1-olakg   1/1     Running   0          1h

# kubectl get svc
NAME            CLUSTER-IP      EXTERNAL-IP   PORT(S)      AGE
kibana-logging   169.169.195.177   <none>        5601/TCP   1h

# kubectl get rc
NAME           DESIRED   CURRENT   AGE
kibana-logging-v1   1         1         1h
```

结果表明运行均已成功。通过 `kubectl cluster-info` 命令获取 Kibana 服务的 URL 地址:

```
# kubectl cluster-info
Kibana is running at http://127.0.0.1:8080/api/v1/proxy/namespaces/kube-system/
services/kibana-logging
```

同样通过 `kubectl proxy` 命令启动代理，在出现 Starting to serve on 127.0.0.1:8001 字样之后，用浏览器访问 URL 地址即可访问 Kibana 页面了：<http://192.168.18.3:8001/api/v1/proxy/namespaces/>

kube-system/services/kibana-logging。

第1次进入页面需要进行一些设置，如图 5.17 所示，选择所需选项后单击 `create`。

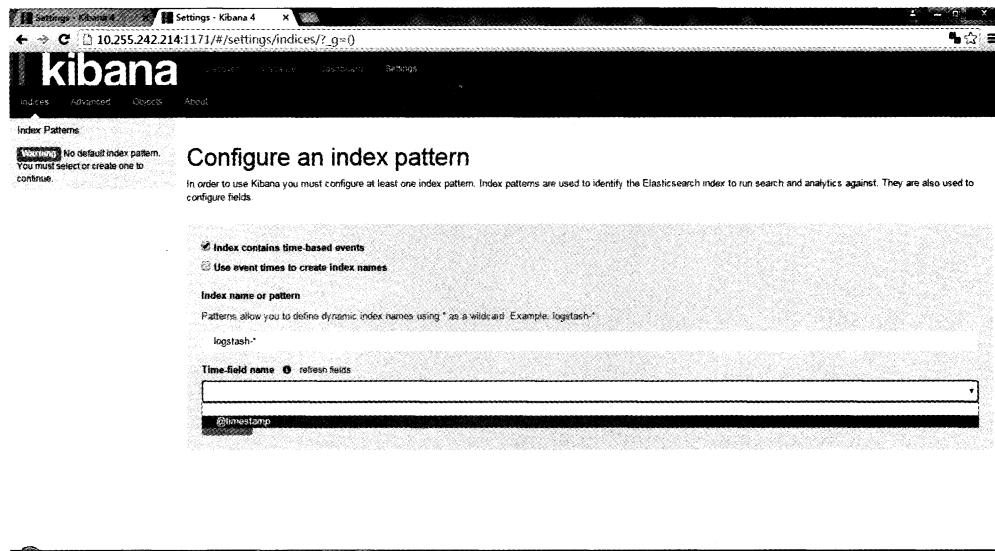


图 5.17 Kibana 创建索引页面

然后单击 discover，就可以正常查询日志了，如图 5.18 所示。

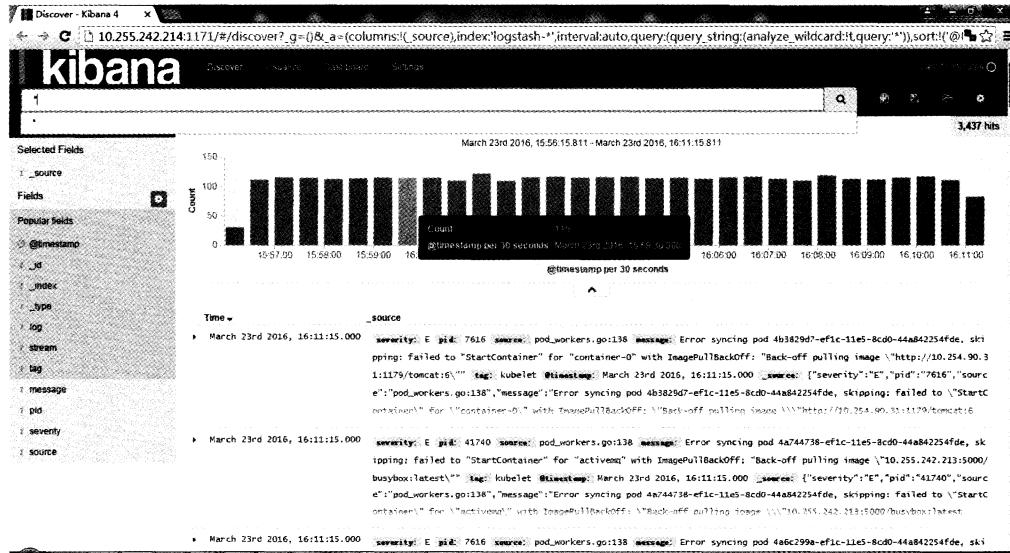


图 5.18 Kibana 查询日志页面

在搜索栏输入“error”关键字，可以搜索出从某些 Node 上找到的日志记录，如图 5.19 所示。

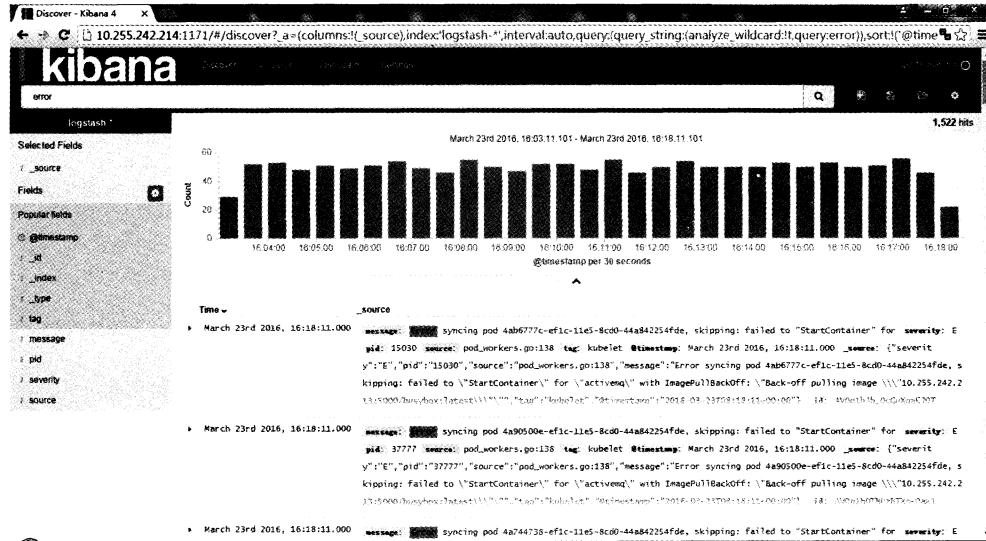


图 5.19 Kibana 日志关键字搜索页面

同时，通过左边菜单中 Fields 相关的内容对查询的内容进行限定，如图 5.20 所示。

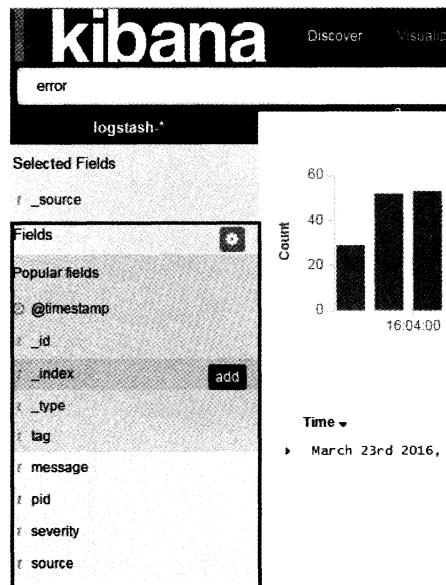


图 5.20 Kibana 查询日志

至此，Kubernetes 集群范围内的统一日志收集和查询系统就搭建完成了。

5.2.2 Cassandra 集群部署案例

Apache Cassandra 是一套开源分布式 NoSQL 数据库系统，其主要特点就是它不是单个数据库，而是由一组数据库节点共同构成的一个分布式的集群数据库。由于 Cassandra 使用的是“去中心化”模式，所以当集群里的一个节点启动之后需要一个途径获知集群中新节点的加入。Cassandra 使用了 Seed（种子）的概念来完成在集群中节点之间的相互查找和通信。

本例通过对 Kubernetes 中 Service 概念的巧妙使用实现了各 Cassandra 节点之间的相互查找。

1. 自定义 SeedProvider

在本例中使用了一个自定义的 SeedProvider 类来完成新节点的查询和添加，类名为 io.k8s.cassandra.KubernetesSeedProvider。

KubernetesSeedProvider.java 类的源代码节选如下：

```
.....
public List<InetAddress> getSeeds() {
    List<InetAddress> list = new ArrayList<InetAddress>();
    String host = "https://kubernetes.default.cluster.local";
    String serviceName = getEnvOrDefault("CASSANDRA_SERVICE", "cassandra");
    String podNamespace = getEnvOrDefault("POD_NAMESPACE", "default");
    String path = String.format("/api/v1/namespaces/%s/endpoints/",
podNamespace);
    .....
    public static void main(String[] args) {
        SeedProvider provider = new KubernetesSeedProvider(new HashMap<String,
String>());
        System.out.println(provider.getSeeds());
    }
}
}
```

完整的源代码可以从这里获取：<http://kubernetes.io/v1.0/examples/cassandra/java/src/io/k8s/cassandra/KubernetesSeedProvider.java>

创建 Cassandra Pod 的配置文件如下：

cassandra.yaml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
```

```

name: cassandra
name: cassandra
spec:
  containers:
    - args:
      - /run.sh
      resources:
        limits:
          cpu: "0.5"
      image: gcr.io/google_containers/cassandra:v5
      name: cassandra
      ports:
        - name: cql
          containerPort: 9042
        - name: thrift
          containerPort: 9160
      volumeMounts:
        - name: data
          mountPath: /cassandra_data
    env:
      - name: MAX_HEAP_SIZE
        value: 512M
      - name: HEAP_NEWSIZE
        value: 100M
      - name: POD_NAMESPACE
        valueFrom:
          fieldRef:
            fieldPath: metadata.namespace
    volumes:
      - name: data
        emptyDir: {}

```

需要说明的是，在镜像 `gcr.io/google_containers/cassandra:v5` 中安装了一个标准的 Cassandra 应用程序，并将定制的 `SeederProvider` 类——`KubernetesSeederProvider` 打包到镜像中了。

定制的 `KubernetesSeederProvider` 类将使用 REST API 来访问 Kubernetes Master，然后通过查询 `name=cassandra` 的服务指向的 Pod 来完成对其他“节点”的查找。

2. 通过 Service 动态查找 Pod

在 `KubernetesSeederProvider` 类中，通过查询环境变量 `CASSANDRA_SERVICE` 的值来获得服务的名称。这样就要求 Service 需要在 Pod 之前创建出来。如果我们已经创建好 DNS 服务（参见 5.1 节的案例介绍），那么也可以直接使用服务的名称而无须使用环境变量。

回顾一下 Service 的概念。Service 通常用作一个负载均衡器，供 Kubernetes 集群中其他应

用（Pod）对属于该 Service 的一组 Pod 进行访问。由于 Pod 的创建和销毁都会实时更新 Service 的 Endpoints 数据，所以可以动态地对 Service 的后端 Pod 进行查询了。Cassandra 的“去中心化”设计使得 Cassandra 集群中的一个 Cassandra 实例（节点）只需要查询到其他节点，即可自动组成一个集群，正好可以使用 Service 的这个特性查询到新增的节点。图 5.21 描述了 Cassandra 新节点加入集群的过程。

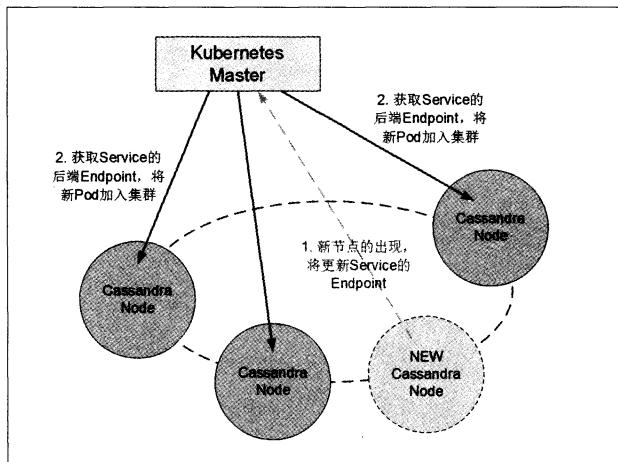


图 5.21 Cassandra 新节点加入集群的过程

在 Kubernetes 系统中，首先需要为 Cassandra 集群定义一个 Service。

cassandra-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: cassandra
  name: cassandra
spec:
  ports:
    - port: 9042
  selector:
    name: cassandra
```

在 Service 的定义中指定 Label Selector 为 name=cassandra。

(1) 创建 Service:

```
$ kubectl create -f cassandra-service.yaml
```

(2) 创建一个 Cassandra Pod:

```
$ kubectl create -f cassandra-pod.yaml
```

现在，一个名为 cassandra 的 Pod 运行起来了，但还没有组成 Cassandra 集群。

(3) 创建一个 RC 来控制 Pod 集群：

cassandra-controller.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: cassandra
  name: cassandra
spec:
  replicas: 1
  selector:
    name: cassandra
  template:
    metadata:
      labels:
        name: cassandra
    spec:
      containers:
        - command:
          - /run.sh
        resources:
          limits:
            cpu: 0.5
      env:
        - name: MAX_HEAP_SIZE
          value: 512M
        - name: HEAP_NEWSIZE
          value: 100M
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      image: gcr.io/google_containers/cassandra:v5
      name: cassandra
      ports:
        - containerPort: 9042
          name: cql
        - containerPort: 9160
          name: thrift
      volumeMounts:
        - mountPath: /cassandra_data
          name: data
    volumes:
      - name: data
```

```
emptyDir: {}
```

由于在 RC 定义中指定的 replicas 数量为 1，所以创建 RC 后，仍然只有之前创建的那个名为 cassandra 的 Pod 在运行。

3. Cassandra 集群中新节点的自动添加

现在，我们使用 Kubernetes 提供的 Scale（动态缩放）机制对 Cassandra 集群进行扩容：

```
$ kubectl scale rc cassandra --replicas=2
```

查看 Pod，可以看到 RC 创建并启动了一个新的 Pod：

```
$ kubectl get pods -l="name=cassandra"
NAME          READY   STATUS    RESTARTS   AGE
cassandra     1/1     Running   0          5m
cassandra-g52t3 1/1     Running   0          50s
```

使用 Cassandra 提供的 nodetool 工具对任一 cassandra 实例（Pod）进行访问来验证 Cassandra 集群的状态。下面的命令将访问名为 cassandra 的 Pod（访问 cassandra-g52t3 也能获得相同的结果）：

```
$ kubectl exec -ti cassandra -- nodetool status
Datacenter: datacenter1
=====
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
-- Address      Load      Tokens  Owns (effective)  Host ID            Rack
UN 10.1.20.16  51.58 KB   256     100.0%           1625c65d-b5b6-40f4-a794-
6f5a12322d86  rack1
UN 10.1.10.11  51.51 KB   256     100.0%           cdfcbf1a-795c-4412-9d3f-
e8fe50bb8deb  rack1
```

可以看到 Cassandra 集群中有两个节点处于正常运行状态（Up and Normal, UN）。结果中的两个 IP 地址为两个 Cassandra Pod 的 IP 地址。

内部的过程为：每个 Cassandra 节点（Pod）通过 API 访问 Kubernetes Master，查询名为 cassandra 的 Service 的 Endpoints（即 Cassandra 节点），若发现有新节点加入，就进行添加操作，最后成功组成了一个 Cassandra 集群。

我们再增加两个 Cassandra 实例：

```
$ kubectl scale rc cassandra --replicas=4
```

用 nodetool 工具查看 Cassandra 集群状态：

```
$ kubectl exec -ti cassandra -- nodetool status
Datacenter: datacenter1
=====
```

```
Status=Up/Down
| / State=Normal/Leaving/Joining/Moving
-- Address      Load      Tokens  Owns (effective)  Host ID          Rack
UN 10.1.20.16  51.58 KB   256     50.5%           1625c65d-b5b6-40f4-a794-
6f5a12322d86  rack1
UN 10.1.10.12  52.03 KB   256     47.0%           8bcc1c3e-44ec-46a7-b981-
4090b206f14e  rack1
UN 10.1.20.17  68.05 KB   256     50.6%           579b6493-e92a-47f5-91f2-
9313198a24c9  rack1
UN 10.1.10.11  51.51 KB   256     51.9%           cdfcbf1a-795c-4412-9d3f-
e8fe50bb8deb  rack1
```

可以看到 4 个 Cassandra 节点都加入 Cassandra 集群中了。

另外，可以通过查看 Cassandra Pod 的日志来看到新节点加入集群的记录：

```
$ kubectl logs cassandra-g52t3
.....
INFO 18:05:36 Handshaking version with /10.1.20.17
INFO 18:05:36 Node /10.1.20.17 is now part of the cluster
INFO 18:05:36 InetSocketAddress /10.1.20.17 is now UP
INFO 18:05:38 Handshaking version with /10.1.10.12
INFO 18:05:39 Node /10.1.10.12 is now part of the cluster
INFO 18:05:39 InetSocketAddress /10.1.10.12 is now UP
```

本例描述了一种通过 API 查询 Service 来完成动态 Pod 发现的应用场景。对于类似于 Cassandra 集群的应用，都可以使用对 Service 进行查询后端 Endpoints 这种巧妙的方法来实现对应用集群（属于同一 Service）中新加入节点的查找。

5.3 Trouble Shooting 指导

本节将对 Kubernetes 集群中常见的问题的排查方法进行说明。

为了跟踪和发现 Kubernetes 集群中运行的容器应用出现的问题，常用的查错方法如下。

首先，查看 Kubernetes 对象的当前运行时信息，特别是与对象关联的 Event 事件。这些事件记录了相关主题、发生时间、最近发生时间、发生次数及事件原因等，对排查故障非常有价值。此外，通过查看对象的运行时数据，我们还可以发现参数错误、关联错误、状态异常等明显问题。由于 Kubernetes 中多种对象相互关联，因此，这一步可能会涉及多个相关对象的排查问题。

其次，对于服务、容器的问题，则可能需要深入容器内部进行故障诊断，此时可以通过查看容器的运行日志来定位具体问题。

最后，对于某些复杂问题，比如 Pod 调度这种全局性的问题，可能需要结合集群中每个节点上的 Kubernetes 服务日志来排查。比如搜集 Master 上 kube-apiserver、kube-schedule、kube-controller-manager 服务的日志，以及各个 Node 节点上的 kubelet、kube-proxy 服务的日志，综合判断各种信息，我们就能找到问题的原因并解决问题。

5.3.1 查看系统 Event 事件

在 Kubernetes 集群中创建了 Pod 之后，我们可以通过 kubectl get pods 命令查看 Pod 列表，但该命令能够显示的信息很有限。Kubernetes 提供了 kubectl describe pod 命令来查看一个 Pod 的详细信息。

```
$ kubectl describe pod redis-master-bobr0
Name:           Redis-master-bobr0
Namespace:      default
Image(s):       kubeguide/Redis-master
Node:          k8s-node-1/192.168.18.3
Labels:         name=Redis-master,role=master
Status:        Running
Reason:
Message:
IP:            172.17.0.58
Replication Controllers:   Redis-master (1/1 replicas created)
Containers:
  master:
    Image:      kubeguide/Redis-master
    Limits:
      cpu:       250m
      memory:    64Mi
    State:      Running
    Started:    Fri, 21 Aug 2015 14:45:37 +0800
    Ready:      True
    Restart Count: 0
  Conditions:
    Type      Status
    Ready     True
  Events:
    FirstSeen     LastSeen     Count   From             SubobjectPath
Reason      Message
Fri, 21 Aug 2015 14:45:36 +0800   Fri, 21 Aug 2015 14:45:36 +0800 1
{kubelet k8s-node-1}  implicitly required container POD pulled          Pod
container image "myregistry:5000/google_containers/pause:latest" already present on
machine
Fri, 21 Aug 2015 14:45:37 +0800   Fri, 21 Aug 2015 14:45:37 +0800 1
```

```
{kubelet k8s-node-1}  implicitly required container POD  created      Created
with docker id a4aa97813908
    Fri, 21 Aug 2015 14:45:37 +0800      Fri, 21 Aug 2015 14:45:37 +0800 1
{kubelet k8s-node-1}  implicitly required container POD  started      Started
with docker id a4aa97813908
    Fri, 21 Aug 2015 14:45:37 +0800      Fri, 21 Aug 2015 14:45:37 +0800 1
{kubelet k8s-node-1}  spec.containers{master}          created
Created with docker id 1e746245f768
    Fri, 21 Aug 2015 14:45:37 +0800      Fri, 21 Aug 2015 14:45:37 +0800 1
{kubelet k8s-node-1}  spec.containers{master}          started
Started with docker id 1e746245f768
    Fri, 21 Aug 2015 14:45:37 +0800      Fri, 21 Aug 2015 14:45:37 +0800 1
{scheduler }           scheduled      Successfully assigned
Redis-master-bobr0 to k8s-node-1
```

该命令除了显示 Pod 创建时的配置定义、状态等信息，还显示了与该 Pod 相关的最近的 Event 事件，事件信息对于查错非常有用。如果某个 Pod 一直处于 Pending 状态，则我们通过 kubectl describe 命令就能了解到失败的具体原因。例如，从 Event 事件中我们可能获知 Pod 失败的原因有以下几种。

- ◎ 没有可用的 Node 以供调度。
- ◎ 开启了资源配置管理并且当前 Pod 的目标节点上恰好没有可用的资源。
- ◎ 正在下载镜像。

kubectl describe 命令还可用于查看其他 Kubernetes 对象，包括 Node、RC、Service、Namespace、Secrets 等，对于每一种对象都会显示相关联的其他信息。

例如，查看一个服务的详细信息：

```
$ kubectl describe service redis-master
Name:           Redis-master
Namespace:      default
Labels:         name=Redis-master
Selector:       name=Redis-master
Type:          ClusterIP
IP:            169.169.208.57
Port:          <unnamed>     6379/TCP
Endpoints:     172.17.0.58:6379
Session Affinity: None
No events.
```

如果查看的对象属于某个特定的 namespace，则需要加上 --namespace=<namespace> 进行查询。例如：

```
$ kubectl get service kube-dns --namespace=kube-system
```

5.3.2 查看容器日志

在需要排查容器内部应用程序生成的日志时，我们可以使用 `kubectl logs <pod_name>` 命令：

```
$ kubectl logs redis-master-bobr0
[1] 21 Aug 06:45:37.781 * Redis 2.8.19 (00000000/0) 64 bit, stand alone mode,
port 6379, pid 1 ready to start.
[1] 21 Aug 06:45:37.781 # Server started, Redis version 2.8.19
[1] 21 Aug 06:45:37.781 # WARNING overcommit_memory is set to 0! Background save
may fail under low memory condition. To fix this issue add 'vm.overcommit_memory =
1' to /etc/sysctl.conf and then reboot or run the command 'sysctl
vm.overcommit_memory=1' for this to take effect.
[1] 21 Aug 06:45:37.782 # WARNING you have Transparent Huge Pages (THP) support
enabled in your kernel. This will create latency and memory usage issues with Redis.
To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/
enabled' as root, and add it to your /etc/ rc.local in order to retain the setting
after a reboot. Redis must be restarted after THP is disabled.
[1] 21 Aug 06:45:37.782 # WARNING: The TCP backlog setting of 511 cannot be enforced
because /proc/sys/net/core/somaxconn is set to the lower value of 128.
```

如果在一个 Pod 中包含多个容器，则需要通过 `-c` 参数指定容器的名称来进行查看，例如：

```
kubectl logs <pod_name> -c <container_name>
```

这个命令与在 Pod 的宿主机上运行 `docker logs <container_id>` 的效果是一样的。

容器中应用程序生成的日志与容器的生命周期是一致的，所以在容器被销毁之后，容器内部的文件也会被丢弃，包括日志等。如果需要保留容器内应用程序生成的日志，则一方面可以使用挂载的 Volume（存储卷）将容器产生的日志保存到宿主机，另一方面也可以通过一些工具对日志进行采集，包括 Fluentd、ElasticSearch 等开源软件。

5.3.3 查看 Kubernetes 服务日志

如果在 Linux 系统上进行安装，并且使用 systemd 系统来管理 Kubernetes 服务，那么 systemd 的 journal 系统会接管服务程序的输出日志。在这种环境中，可以通过使用 `systemctl status` 或 `journalctl` 工具来查看系统服务的日志。

例如，使用 `systemctl status` 命令查看 `kube-controller-manager` 服务的日志：

```
# systemctl status kube-controller-manager -l
kube-controller-manager.service - Kubernetes Controller Manager
   Loaded: loaded (/usr/lib/systemd/system/kube-controller-manager.service;
   enabled)
     Active: active (running) since Fri 2015-08-21 18:36:29 CST; 5min ago
       Docs: https://github.com/GoogleCloudPlatform/kubernetes
      Main PID: 20339 (kube-controller)
         Tasks: 1 (who: /usr/bin/kube-controller-manager --v=2 --log-dir=/var/log/kube-
```

```
CGroup: /system.slice/kube-controller-manager.service
└─20339 /usr/bin/kube-controller-manager --logtostderr=false --v=4
--master=http://kubernetes-master:8080 --log_dir=/var/log/kubernetes

Aug 21 18:36:29 kubernetes-master systemd[1]: Starting Kubernetes Controller Manager...
Aug 21 18:36:29 kubernetes-master systemd[1]: Started Kubernetes Controller Manager.
```

使用 journalctl 命令查看：

```
# journalctl -u kube-controller-manager
-- Logs begin at Mon 2015-08-17 16:43:22 CST, end at Fri 2015-08-21 18:36:29 CST.
--
Aug 17 16:44:14 kubernetes-master systemd[1]: Starting Kubernetes Controller Manager...
Aug 17 16:44:14 kubernetes-master systemd[1]: Started Kubernetes Controller Manager.
```

如果不使用 systemd 系统接管 Kubernetes 服务的标准输出，则也可以通过日志相关的启动参数来指定日志的存放目录。

- ◎ `--logtostderr=false`: 不输出到 stderr。
- ◎ `-log-dir=/var/log/kubernetes`: 日志的存放目录。
- ◎ `--alsologtostderr=false`: 设置为 true 则表示将日志输出到文件时也输出到 stderr。
- ◎ `--v=0`: glog 日志级别。
- ◎ `--vmodule=gfs*=2,test*=4`: glog 基于模块的详细日志级别。

在`--log_dir` 设置的目录中可以查看各服务进程生成的日志文件，日志文件的数量和大小依赖于日志级别的设置。例如 `kube-controller-manager` 可能生成的几个日志文件如下。

- ◎ `kube-controller-manager.ERROR`。
- ◎ `kube-controller-manager.INFO`。
- ◎ `kube-controller-manager.WARNING`。
- ◎ `kube-controller-manager.kubernetes-master.unknownuser.log.ERROR.20150930-173939.9847`。
- ◎ `kube-controller-manager.kubernetes-master.unknownuser.log.INFO.20150930-173939.9847`。
- ◎ `kube-controller-manager.kubernetes-master.unknownuser.log.WARNING.20150930-173939.9847`。

在大多数情况下，我们从 WARNING 和 ERROR 级别的日志中就能找到问题的原因，但有时还是需要排查 INFO 级别的日志甚至 DEBUG 级别的详细日志。此外，etcd 服务也属于

Kubernetes 集群中的重要组成部分，所以它的日志也不能忽略。

如果是某个 Kubernetes 对象存在问题，则我们可以用这个对象的名字作为关键字搜索 Kubernetes 的日志来发现和解决问题。在大多数情况下，我们平常所遇到的主要是与 Pod 对象相关的问题，比如无法创建 Pod、Pod 启动后就停止或者 Pod 副本无法增加等。此时，我们可以先确定 Pod 在哪个节点上，然后登录这个节点，从 kubelet 的日志中查询该 Pod 的完整日志，然后进行问题排查。对于与 Pod 扩容相关或者与 RC 相关的问题，则很可能在 kube-controller-manager 及 kube-scheduler 的日志上找出问题的关键点。

另外，kube-proxy 经常被我们忽视，因为即使它意外地被停止，Pod 的状态也是正常的，但会导致某些服务访问异常的情况。这些错误通常与每个节点上的 kube-proxy 服务有着密切的关系。遇到这些问题时，首先要排查 kube-proxy 服务的日志，同时排查防火墙服务，特别是要留意防火墙中是否有人为添加的可疑规则。

5.3.4 常见问题

本节对 Kubernetes 系统中的一些常见问题及解决方法进行说明。

1. 由于无法下载 pause 镜像导致 Pod 一直处于 Pending 的状态

以 redis-master 为例，使用如下配置文件 redis-master-controller.yaml 创建 RC 和 Pod：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-master
  labels:
    name: redis-master
spec:
  replicas: 1
  selector:
    name: redis-master
  template:
    metadata:
      labels:
        name: redis-master
    spec:
      containers:
        - name: master
          image: kubeguide/redis-master
          ports:
            - containerPort: 6379
```

执行 `kubectl create -f redis-master-controller.yaml` 成功。

但在查看 Pod 时，发现其总是无法处于 Running 状态。通过 `kubectl get pods` 命令可以看到：

```
$ kubectl get pods
NAME          READY   STATUS             RESTARTS   AGE
redis-master-6yy7o   0/1    Image: kubeguide/redis-master is ready, container
is creating     0        5m
```

进一步使用 `kubectl describe pod redis-master-6yy7o` 命令查看该 Pod 的详细信息：

```
$ kubectl describe pod redis-master-6yy7o
Name:           redis-master-6yy7o
Namespace:      default
Image(s):       kubeguide/redis-master
Node:           127.0.0.1/127.0.0.1
Labels:          name=redis-master
Status:          Pending
Reason:
Message:
IP:
Replication Controllers:      redis-master (1/1 replicas created)
Containers:
  master:
    Image:       kubeguide/redis-master
    State:       Waiting
    Reason:      Image: kubeguide/redis-master is ready, container is
creating
    Ready:       False
    Restart Count: 0
  Conditions:
    Type     Status
    Ready   False
  Events:
    FirstSeen   LastSeen   Count   From   SubobjectPath
Reason   Message
Thu, 24 Sep 2015 19:19:25 +0800   Thu, 24 Sep 2015 19:25:58 +0800  3
{kubelet 127.0.0.1}   failedSync Error syncing pod, skipping: image pull failed
for gcr.io/google_containers/pause-amd64:3.0, this may be because there are no
credentials on this request. details: (API error (500): invalid registry endpoint
https://gcr.io/v0/: unable to ping registry endpoint https://gcr.io/v0/v2 ping
attempt failed with error: Get https://gcr.io/v2/: dial tcp 173.194.196.82:443:
connection refused v1 ping attempt failed with error: Get https://gcr.io/v1/_ping:
dial tcp 173.194.79.82:443: connection refused. If this private registry supports
only HTTP or HTTPS with an unknown CA certificate, please add '--insecure-registry
gcr.io` to the daemon's arguments. In the case of HTTPS, if you have access to the
registry's CA certificate, no need for the flag; simply place the CA certificate at
/etc/docker/certs.d/gcr.io/ca.crt)
```

```

Thu, 24 Sep 2015 19:19:25 +0800      Thu, 24 Sep 2015 19:25:58 +0800 3
{kubelet 127.0.0.1}  implicitly required container POD failed Failed to pull
image "gcr.io/google_containers/pause-amd64:3.0": image pull failed for
gcr.io/google_containers/pause:0.8.0, this may be because there are no credentials
on this request. details: (API error (500): invalid registry endpoint https://
gcr.io/v0/: unable to ping registry endpoint https://gcr.io/v0/v2 ping attempt failed
with error: Get https://gcr.io/v2/: dial tcp 173.194.196.82:443: connection refused
v1 ping attempt failed with error: Get https://gcr.io/v1/_ping: dial tcp 173.194.79.82:
443: connection refused. If this private registry supports only HTTP or HTTPS with
an unknown CA certificate, please add `--insecure-registry gcr.io` to the daemon's
arguments. In the case of HTTPS, if you have access to the registry's CA certificate,
no need for the flag; simply place the CA certificate at
/etc/docker/certs.d/gcr.io/ca.crt

```

可以看到，该 Pod 的状态为 Pending，从 Message 部分显示的信息可以看出其原因是 image pull failed for gcr.io/google_containers/pause-amd64:3.0，说明系统在创建 Pod 时无法从 gcr.io 下载 pause 镜像，所以导致创建 Pod 失败。

解决方法如下。

(1) 如果服务器可以访问 Internet，并且不希望使用 HTTPS 的安全机制来访问 gcr.io，则可以在 Docker Daemon 的启动参数中加上--insecure-registry gcr.io 来表示可以进行匿名下载。

(2) 如果 Kubernetes 集群环境在内网环境中，无法访问 gcr.io 网站，则可以先通过一台能够访问 gcr.io 的机器将 pause 镜像下载下来，导出后，再导入内网的 Docker 私有镜像库中，并在 kubelet 的启动参数中加上--pod_infra_container_image，配置为：

```
--pod_infra_container_image=<docker_registry_ip>:<port>/google_containers/pa
use-amd64:3.0
```

之后重新创建 redis-master 即可正确启动 Pod 了。

注意，除了 pause 镜像，其他 Docker 镜像也可能存在无法下载的情况，与上述情况类似，很可能也是网络配置使得镜像无法下载，解决方法同上。

2. Pod 创建成功，但状态始终不是 Ready，且 RESTARTS 的数量持续增加

在创建了一个 RC 之后，通过 kubectl get pods 命令查看 Pod，发现如下情况：

```

.....
$ kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
zk-bg-ri3ru  0/1      Running   3          37s
.....
$ kubectl get pods
NAME        READY     STATUS    RESTARTS   AGE
zk-bg-ri3ru  0/1      Running   5          1m

```

```
.....
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
zk-bg-ri3ru  0/1     ExitCode:0   6          1m
.....
$ kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
zk-bg-ri3ru  0/1     Running   7          1m
```

可以看到 Pod 已经创建成功了，但 Pod 的状态一会儿是 Running，一会儿是 ExitCode:0，READY 列中始终无法变成 1，而且 RESTARTS（重启的数量）的数量不断增加。

通常造成这种现象是因为容器的启动命令不能保持前台运行。

本例中的 Docker 镜像的启动命令为：

```
zkServer.sh start-background
```

在 Kubernetes 根据 RC 定义创建 Pod 后启动容器，容器的启动命令执行完成时，即认为该容器的运行已经结束，并且是成功结束（ExitCode=0）。然后，根据 Pod 的默认重启策略定义（RestartPolicy=Always），RC 将启动这个容器。

新的容器执行启动命令后仍然会成功结束，然后 RC 会再次重启该容器，进入一个无限循环的过程中。

解决方法为将 Docker 镜像的启动命令设置为一个前台运行的命令，例如：

```
zkServer.sh start-foreground
```

5.3.5 寻求帮助

如果通过系统日志和容器日志都无法找到出现问题的原因，则还可以追踪源码进行分析，或者通过一些在线途径寻求帮助。

- ◎ Kubernetes 的常见问题参见 <https://github.com/GoogleCloudPlatform/kubernetes/wiki/User-FAQ>。
- ◎ Debugging 的常见问题参见 <https://github.com/GoogleCloudPlatform/kubernetes/wiki/Debugging-FAQ>。
- ◎ Service 的常见问题参见 <https://github.com/GoogleCloudPlatform/kubernetes/wiki/Services-FAQ>。
- ◎ StackOverflow 网站关于 Kubernetes 的主题参见 <http://stackoverflow.com/questions/tagged/kubernetes> 或 <http://stackoverflow.com/questions/tagged/google-container-engine>。
- ◎ IRC 频道（#google-containers）参见 <https://botbot.me/freenode/google-containers/>。
- ◎ Kubernetes 邮件列表 Email 参见 google-containers@googlegroups.com。

5.4 Kubernetes v1.3 开发中的新功能

本节对 Kubernetes v1.3 版本的正在开发中的一些新功能进行介绍，包括 Pet Set（用于管理有状态的容器应用）、init container（Pod 中的初始化容器）、Cluster Federation（集群联邦管理）等内容。

5.4.1 Pet Set（有状态的容器）

在 Kubernetes 集群中，组成一个微服务的后端 Pod 一般来说都是无状态的容器应用。例如通过 RC 来进行管理，只需要维持 Pod 的副本数量即可，当某个 Pod 失败时就直接销毁并重新创建一个新的 Pod，提供能够水平扩展的微服务。我们可以称这类应用为“Cattle”（农场动物），即单个实例不是特别重要，可随时被替换。

但对于有状态的应用来说，即以集群的方式部署的大型应用软件，每个实例都需要具备唯一的标识，并且各个实例可能还有启动顺序的要求。v1.3 版本新增了一种名为 PetSet 的资源对象，用于支持有状态的容器应用。我们可以称这类应用为“Pet”（宠物），即每个实例都非常重要，其身份不应被别的实例替换。

Pet Set 能够确保为每个 Pet 设置一个唯一的身份标识，包括如下几种。

- ◎ 唯一且不变的 hostname，并保存在 DNS 中。
- ◎ 唯一的顺序编号，用于确保各实例的启动顺序。
- ◎ 为每个容器提供永久存储，与其 hostname 和启动顺序绑定。

Pet Set 能够用于许多应用场景，如下所述。

- ◎ 数据库应用，例如 MySQL 或 PostgreSQL，其每个实例都需要挂载一个外部的永久存储。
- ◎ 集群化的应用软件，例如 ZooKeeper、etcd、ElasticSearch 等需要集群中的各成员有稳定的身份。

下面的例子描述了 PetSet 的创建和用法。

```
petset.yaml
# 使用 headless Service，以创建相应 Pod 的 DNS 记录
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
```

```
spec:
  ports:
    - port: 80
      name: web
    # *.nginx.default.svc.cluster.local
  clusterIP: None
  selector:
    app: nginx
---
# PetSet 的定义
apiVersion: apps/v1alpha1
kind: PetSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        pod.alpha.kubernetes.io/initialized: "true"
    spec:
      terminationGracePeriodSeconds: 0
      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
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

在 PetSet 定义中，需要设置永久存储“volumeClaimTemplates”，需要系统管理员预先创建好外部 PV（Persistent Volume），才能给 PetSet 使用。

使用 kubectl create 命令创建该 PetSet:

```
$ kubectl create -f petset.yaml
service "nginx" created
petset "nginx" created
```

查看创建好的 Pod 的信息:

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1      Running   0          10m
web-1     1/1      Running   0          10m
```

可以看到每个 Pod 的名称不再是通过 RC 创建的带有一个 UUID 的名称，而是由“`petset name`”与一个序列号组成的特定名称：`web-0` 和 `web-1`。

而 Pod 名称也就是该 Pod 的 hostname，即 Pet Set 为每个 Pet 设置了稳定的主机名。

```
# kubectl exec web-0 -- sh -c 'hostname'
web-0
# kubectl exec web-1 -- sh -c 'hostname'
web-1
```

同时，每个 Pod 的网络身份也通过 Service 的定义被创建出来。根据 Service 的定义，该 Service 将在 DNS 中生成一条没有 ClusterIP 的记录：

`nginx.default.svc.cluster.local`

该 Service 的后端为两个 Pet 的地址（可以看成是 Pet 的服务名）：

```
web-0.nginx.default.svc.cluster.local
web-1.nginx.default.svc.cluster.local
```

这两个 Pet 的地址 `web-0.nginx` 和 `web-1.nginx` 将作为每个 Pet 的稳定网络身份被系统保存在 DNS 中，供客户端应用访问。

接下来通过一个 busybox 容器执行 nslookup，验证 Pet 的服务地址：

```
# kubectl run -i --tty --image busybox dns-test --restart=Never /bin/sh
Hit enter for command prompt
```

查询 `web-0.nginx`:

```
/ # nslookup web-0.nginx
Server:  169.169.0.100
Address 1: 169.169.0.100

Name:      web-0.nginx
Address 1: 172.17.1.2
```

查询 `web-1.nginx`:

```
/ # nslookup web-1.nginx
```

```
Server: 169.169.0.100
Address 1: 169.169.0.100
```

```
Name: web-1.nginx
Address 1: 172.17.1.3
```

如果直接查询 Nginx 服务（headless service），则系统将返回后端两个 Pet 的 IP 地址列表：

```
/ # nslookup nginx
Server: 169.169.0.100
Address 1: 169.169.0.100
```

```
Name: nginx
Address 1: 172.17.1.2
Address 2: 172.17.1.3
```

借助于 headless service 的功能，可以实现各 Pet 相互之间进行发现和访问。

当前版本 Pet Set 的使用限制如下。

- ◎ 只有 replicas 字段可以被更新，但更新后 Pet Set 仍然是按顺序依次创建各 Pet。
- ◎ 删除 petset 时，系统不会自动删除已经运行中的 Pets，需要手工删除。
- ◎ 出于数据安全的考虑，在删除 Pet 时系统不会自动删除该 Pet 使用的 PV 存储。
- ◎ 目前不支持 Pet 镜像的滚动升级操作，需要手工完成。

5.4.2 Init Container（初始化容器）

在很多应用场景中，应用在启动之前都需要一些初始化的操作，例如：

- ◎ 等待其他关联组件正确运行（例如数据库或某个后台服务）；
- ◎ 基于环境变量或配置模板生成配置文件；
- ◎ 从远程数据库获取本地所需配置，或者将自身注册到某个中央数据库；
- ◎ 下载相关依赖包，或者对系统进行一些预配置操作。

Kubernetes v1.3 版本引入了一个 Alpha 版本的新特性：init container，用于在启动普通容器之前启动一个或多个“初始化”容器，完成普通容器所需要的前置条件，如图 5.22 所示。init container 与普通容器本质上是一样的，但它们是仅运行一次就结束的任务，并且必须成功执行完成后，系统才能继续执行下一个容器。根据 Pod 的重启策略（RestartPolicy），当 init container 执行失败，在设置了 RestartPolicy=Never 时，Pod 将会启动失败；而设置 RestartPolicy=Always 时，Pod 将会被系统自动重启。

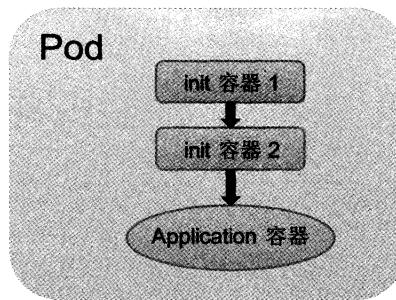


图 5.22 init container

在当前的版本中要启用 init container 的配置，需要在 Pod 的 annotation 字段中设置 pod.alpha.kubernetes.io/init-containers 来定义需要执行的初始化容器列表。

下面，以 Nginx 应用为例，在启动 Nginx 之前，通过初始化容器 busybox 为 Nginx 创建一个 index.html 主页文件。

```

nginx-init-containers.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  annotations:
    pod.alpha.kubernetes.io/init-containers: '[{"name": "install", "image": "busybox", "command": ["wget", "-O", "/work-dir/index.html", "http://kubernetes.io/index.html"], "volumeMounts": [{"name": "workdir", "mountPath": "/work-dir"}]}]'
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: workdir
      mountPath: /usr/share/nginx/html

```

```
dnsPolicy: Default
volumes:
- name: workdir
  emptyDir: {}
```

创建这个 Pod:

```
# kubectl create -f nginx-init-containers.yaml
pod "nginx" created
```

在运行 init container 的过程中，查看 Pod 的状态，可见 Init 过程还未完成：

```
# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
nginx    0/1       Init:0/1   0          1m
```

当 init container 成功执行完成，系统继续启动 Nginx 容器，再次查看 Pod 的状态：

```
# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
nginx    1/1       Running   0          7s
```

查看 Pod 的事件，可以看到系统按顺序运行 Pod 中的各个容器：

```
# kubectl describe pod nginx
Name:           nginx
Namespace:      default
..... (略)
Events:
FirstSeen     LastSeen      Count  From                    SubobjectPath
Type          Reason        Message
-----  -----  -----
4s            4s           1      {default-scheduler } 
Normal        Scheduled    Successfully assigned nginx to k8s-node-1
4s            4s           1      {kubelet k8s-node-1}
spec.initContainers{install}  Normal        Pulled      Container image "busybox"
already present on machine
4s            4s           1      {kubelet k8s-node-1}
spec.initContainers{install}  Normal        Created     Created container with
docker id 81d3ef7ade94
4s            4s           1      {kubelet k8s-node-1}
spec.initContainers{install}  Normal        Started    Started container with
docker id 81d3ef7ade94
3s            3s           1      {kubelet k8s-node-1}
spec.containers{nginx}       Normal        Pulled      Container image "nginx"
already present on machine
3s            3s           1      {kubelet k8s-node-1}
spec.containers{nginx}       Normal        Created     Created container with
docker id 5a0bc53661f6
2s            2s           1      {kubelet k8s-node-1}
```

```
spec.containers{nginx}          Normal        Started      Started container with
docker id 5a0bc53661f6
```

在 init container 的后续演进中，将进一步考虑 Pod 的资源使用限制、健康检查、镜像更新等问题。

5.4.3 Cluster Federation（集群联邦）

集群联邦是管理多个 Kubernetes 集群的集群，用于多个集群中的服务统一管理，以及当一个集群发生故障时，能够将业务恢复到其他集群上，如图 5.23 所示。目前集群联邦只能在谷歌或亚马逊的公有云上进行配置和使用。

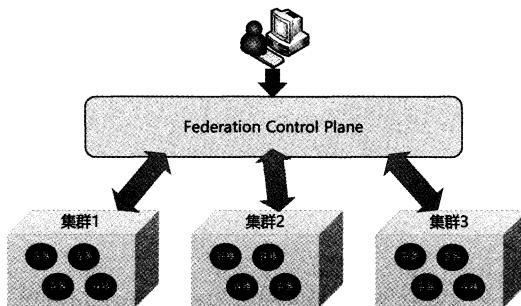


图 5.23 集群联邦

集群联邦的主要组件由 Federation Control Plane（控制平面）来完成对多个集群的管控，其核心组件包括 federation-apiserver 和 federation-controller-manager 和 etcd，可以在已经存在的一个 Kubernetes 集群上以 Pod 的形式启动这些 Federation 组件。

下面以在 GCE 上运行一个 Cluster Federation 为例，创建 federation-apiserver 和 federation-controller-manager 的命令为：

```
$ KUBERNETES_PROVIDER=gce FEDERATION_DNS_PROVIDER=google-clouddns
FEDERATION_NAME=myfederation DNS_ZONE_NAME=myfederation.example
FEDERATION_PUSH_REPO_BASE=gcr.io/google_containers ./federation/cluster/federati
on-up.sh
```

各个参数的含义如下。

- ◎ **KUBERNETES_PROVIDER:** 云服务商。
- ◎ **FEDERATION_DNS_PROVIDER:** 可以是 google-clouddns 或者 aws-route53。如果已经把 KUBERNETES_PROVIDER 设置为 gce、gke 及 aws 中的一个，那么系统会自动设置这一变量。该设置项用于为联邦服务提供域名解析能力。当联邦中的 Kubernetes 集群上的 Pod 或者 Service 发生变更的时候，会在 DNS 记录上做出相应的变更。

- ◎ **FEDERATION_NAME**: 联邦的名称，这一名称也会反映在 DNS 记录之中。
- ◎ **DNS_ZONE_NAME**: DNS 记录的域名。用户需要购买和使用这个域名，让 **FEDERATION_DNS_PROVIDER** 能够为这一域名的查询提供正确的解析结果。

通过上面的命令会创建一个 `federation` 命名空间，并创建两个 `Deployment` 对象：`federation-apiserver` 及 `federation-controller-manager`。

验证创建出来的 `deployment`:

```
$ kubectl get deployments --namespace=federation
NAME             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
federation-apiserver   1         1         1           1          1m
federation-controller-manager 1         1         1           1          1m
```

`federation-up.sh` 还会在 `kubeconfig` 中创建一个新纪录，用来和联邦 `APIServer` 进行通信。可以使用 `kubectl config view` 来查看。

另外，`federation-up.sh` 创建的 `federation-apiserver` Pod 中包含的 `etcd` 容器使用了 PV 持久卷，用来提供持久化数据存储，目前只能在 AWS、GKE 或 GCE 环境中进行创建。具体的 PV 设置可以通过修改 `federation/manifests/federation-apiserver-deployment.yaml` 来完成。

在联邦控制平面启动之后，就可以对现有的 Kubernetes 集群进行纳管了。

首先，我们需要创建一个 `secret` 对象，其中包含了 Kubernetes 集群的 `kubeconfig`，联邦将会用这一内容和受管集群进行通信。假设 `kubeconfig` 文件位于 `/cluster1/kubeconfig`，用下面的命令来创建 `secret`:

```
$ kubectl create secret generic cluster1 --namespace=federation --from-file=/cluster1/kubeconfig
```

文件名 `kubeconfig` 将用于设置 `secret` 的 key 名称。

创建好 `secret` 之后，就可以注册集群了，一个集群对象的 `yaml` 配置文件如下:

```
apiVersion: federation/v1beta1
kind: Cluster
metadata:
  name: cluster1
spec:
  serverAddressByClientCIDRs:
    - clientCIDR: <client-cidr>
      serverAddress: <apiserver-address>
    secretRef:
      name: <secret-name>
```

需要把 `<client-cidr>`、`<apiserver-address>` 及 `<secret-name>` 替换为实际内容。`<secret-name>` 是前面刚刚创建的 `secret` 的名称。`serverAddressByClientCIDRs` 包含一系列地址，符合 CIDR 的客

客户端才能连接服务器的这一地址。我们可以设置服务器地址的 CIDR 为“0.0.0.0/0”，这样所有的客户端都可以访问。另外，如果希望内部客户端使用服务器的 clusterIP，则可以把这一 IP 设置为 serverAddress，然后设置 clientCIDR 为集群内的 Pod 地址范围。

将该 yaml 文件保存为/cluster1/cluster.yaml，运行下面的命令来进行集群的纳管：

```
$ kubectl create -f /cluster1/cluster.yaml --context=federation-cluster
```

设置--context=federation-cluster 意思是将请求发往联邦的 federation-apiserver。

查看纳管的结果：

```
$ kubectl get clusters --context=federation-cluster
NAME      STATUS  VERSION  AGE
cluster1  Ready           3m
```

当集群纳管之后，就可以使用集群联邦的功能了。

为了支持跨集群的服务发现机制，需要扩展 KubeDNS 服务，通过--federations 参数设置集群联邦的总 DNS 服务：

```
--federations=${FEDERATION_NAME}=${DNS_DOMAIN_NAME}
```

可以通过编辑现有 KubeDNS 的 RC 中所包含的 Pod 模板来为 Pod 加入这一参数，删除当前运行的 Pod 之后，RC 就会根据新的模板创建带有联邦 DNS 信息的 KubeDNS 服务了。

```
$ kubectl get rc --namespace=kube-system
```

kube-dns 的 RC 名是 kube-dns-[id]的形式，用 edit 进行编辑：

```
$ kubectl edit rc <rc-name> --namespace=kube-system
```

在弹出的 yaml 文件中加入--federation 参数，保存退出，然后查询并删除现有的 Pod。

```
$ kubectl delete pods <pod-name> --namespace=kube-system
```

至此，集群联邦设置完成，接下来就可以在多个集群上部署应用了。

假设当前已有 4 个集群纳管进了集群联邦：

```
$ kubectl get clusters --context=federation-cluster
NAME      STATUS  VERSION  AGE
cluster1  Ready           3m
cluster2  Ready           3m
cluster3  Ready           3m
cluster4  Ready           3m
```

在这 4 个集群上创建 Nginx 服务：

```
$ kubectl --context=federation-cluster create -f services/nginx.yaml
```

等待该服务在所有集群上创建完成，并且集群联邦内的服务也更新完成，通常这需要几分钟。

查看服务的状态，可以看到在每个集群上创建的 Loadbalancer 的 IP 地址：

```
$ kubectl --context=federation-cluster describe services nginx
Name:           nginx
Namespace:      default
Labels:         run=nginx
Selector:       run=nginx
Type:          LoadBalancer
IP:
LoadBalancer Ingress:   104.197.246.190, 130.211.57.243, 104.196.14.231,
104.199.136.89
Port:          http    80/TCP
Endpoints:     <none>
Session Affinity: None
No events.
```

登录其中一个集群，查看由 federation-controller-manager 创建的 Nginx 服务：

```
$ kubectl --context=cluster1 get svc nginx
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
nginx    10.63.250.98    104.199.136.89    80/TCP      9m
```

可以看到集群联邦通过在每个集群上创建一个同名的服务，但此时由于每个集群的 Service 后端还没有运行任何 Pod，所以这些联邦服务还无法正常工作。

接下来通过在每个集群上运行 Pod 来支撑 Service：

```
$ kubectl --context=cluster1 run nginx --image=nginx:1.11.1-alpine --port=80
$ kubectl --context=cluster2 run nginx --image=nginx:1.11.1-alpine --port=80
$ kubectl --context=cluster3 run nginx --image=nginx:1.11.1-alpine --port=80
$ kubectl --context=cluster4 run nginx --image=nginx:1.11.1-alpine --port=80
```

当这些 Pod 成功运行后，Service 将被集群联邦设置为正常状态，然后集群联邦将会配置相应的公共 DNS 记录。假设使用的是 Google Cloud DNS，并且 DNS 域名为 example.com，则可以看到每个集群上的 Nginx 服务都被设置好了一个 DNS 名，可供其他应用访问时使用：

```
$ gcloud dns managed-zones describe example-dot-com
creationTime: '2016-06-26T18:18:39.229Z'
description: Example domain for Kubernetes Cluster Federation
dnsName: example.com.
id: '3229332181334243121'
kind: dns#managedZone
name: example-dot-com
nameServers:
- ns-cloud-a1.googledomains.com.
- ns-cloud-a2.googledomains.com.
- ns-cloud-a3.googledomains.com.
- ns-cloud-a4.googledomains.com.
```

```
$ gcloud dns record-sets list --zone example-dot-com
NAME                           TYPE    TTL   DATA
example.com.                   NS      21600
ns-cloud-e1.googledomains.com., ns-cloud-e2.googledomains.com.
example.com.                   SOA      21600
ns-cloud-e1.googledomains.com. cloud-dns-hostmaster.google.com. 1 21600 3600
1209600 300
nginx.mynamespace.myfederation.svc.example.com.          A      180
104.XXX.XXX.XXX, 130.XXX.XX.XXX, 104.XXX.XX.XXX, 104.XXX.XXX.XX
nginx.mynamespace.myfederation.svc.cluster1.example.com.  A      180
104.XXX.XXX.XXX
nginx.mynamespace.myfederation.svc.cluster2.example.com.  A      180
104.XXX.XXX.XXX, 104.XXX.XXX.XXX, 104.XXX.XXX.XXX
nginx.mynamespace.myfederation.svc.cluster3.example.com.  A      180
130.XXX.XX.XXX
nginx.mynamespace.myfederation.svc.cluster4.example.com.  A      180
130.XXX.XX.XXX, 130.XXX.XX.XXX
nginx.mynamespace.myfederation.svc.cluster4.example.com.  CNAME   180
nginx.mynamespace.myfederation.svc.example.com.
....
```

第 6 章

Kubernetes 源码导读

6.1 Kubernetes 源码结构和编译步骤

Kubernetes 的源码现在托管在 GitHub 上, 地址为 <https://github.com/googlecloudplatform/kubernetes>。

编译脚本存放在 build 子目录下, 在 Linux 环境(可以是虚拟机)中执行如下命令即可完成代码的编译过程:

```
git clone https://github.com/GoogleCloudPlatform/kubernetes.git  
cd kubernetes/build  
. ./release.sh
```

制作 release 的过程其实有不少有意思的事情发生, 包括启动 Docker 容器来安装 Go 语言环境、etcd 等, 读者若有兴趣则可以查看 release.sh 脚本。另外, 如果编译环境是通过 HTTP 代理上网的, 则需要设置好 Git 与 Docker 相关的 HTTP 代理参数, 同时在文件 kubernetes/build/build-image/Dockerfile 中增加如下 HTTP 代理参数。

- ◎ ENV http_proxy=http://username:password@proxyaddr:proxyport。
- ◎ ENV https_proxy=http://username:password@proxyaddr:proxyport。

在编译过程中产生的与 Docker 相关的 docker image、dockerfile 及编译好的二进制文件包, 则存放在 kubernetes/_output 目录下, 这个目录总共有 4 个子目录: dockerized、images、release-stage、release-tars, 我们关心后两个目录, 其中 release-stage 目录下存放的是支持 linux-amd64 架构的 Server 端的二进制可执行文件(放在 server 子目录下), 以及支持不同平台的 Client 端的二进制可执行文件(放在 client 子目录下), release-tars 则存放的是 release-stage 目录下各级子目录的压缩包, 与从官方网站下载的完全一样。

考虑到学习和调试 Kubernetes 代码的便利性，我们接下来介绍如何在 Windows 的 LiteIDE 开发环境中完成 Kubernetes 代码的编译和调试。本文假设 Windows 上的 GO 运行时框架和 LiteIDE 开发环境已经建立好，并通过 `git clone` 命令已经将 <https://github.com/GoogleCloudPlatform/kubernetes.git> 下载到本地 C:\kubernetes 目录中。通过分析 Kubernetes 的目录结构，我们发现 Kubernetes 的源码都在 `pkg` 子目录下。接下来建立 k8s 工程目录，目录位置为 C:\project\go\k8s，并在里面建立 `src`、`pkg` 两个子目录，然后把 C:\kubernetes\Godeps_workspace\src 全部转移到 C:\project\go\k8s\src 目录下，因为这里是 Kubernetes 源码的所有依赖包，所以如果手动一个一个地下载，则恐怕以国内的网速一天也搞不定。转移完成后，C:\project\go\k8s\src 的目录结构包括如下内容：

```
C:\project\go\k8s\src>dir
2015-07-14 11:56    <DIR>          bitbucket.org
2015-07-14 11:56    <DIR>          code.google.com
2015-07-17 12:30    <DIR>          github.com
2015-07-14 11:56    <DIR>          golang.org
2015-07-14 11:56    <DIR>          google.golang.org
2015-07-14 11:56    <DIR>          gopkg.in
2015-07-14 11:56    <DIR>          speter.net
```

接下来把 C:\kubernetes 的整个目录移动到 C:\project\go\k8s\src\github.com\GoogleCloudPlatform\下，因为 Kubernetes 的源码包的完整名字为“github.com/GoogleCloudPlatform/kubernetes/pkg”。上述工作完成以后，所有的源码都在 C:\project\go\k8s\src 目录下了，我们用 LiteIDE 打开 C:\project\go\k8s，单击菜单“查看”→“管理 Gopath”→添加目录“C:\project\go\k8s”，然后可以进入目录 `github.com/GoogleCloudPlatform/kubernetes/pkg` 下，逐一编译每个 package 目录了，如图 6.1 所示。

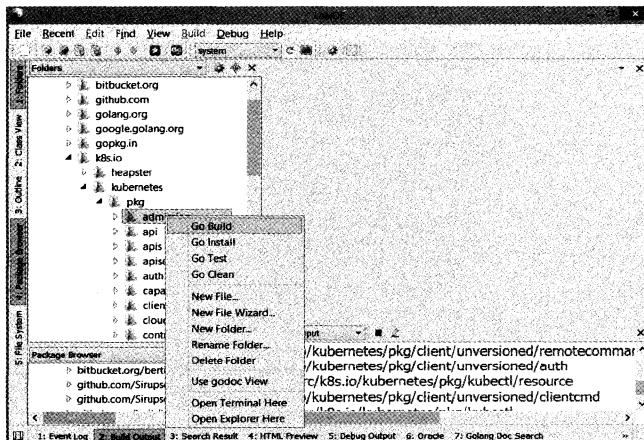


图 6.1 LiteIDE 编译 Kubernetes 的 package

在每个 package 都编译完成以后，我们可以尝试启动 kube-scheduler 进程：在 LiteIDE 里打开 `github.com/GoogleCloudPlatform/kubernetes/pkg/plugin/cmd/kube-scheduler/scheduler.go`，并且按快捷键 Ctrl+R，你会惊奇地发现这个 Kubernetes 服务器端进程竟然也能在 Windows 下运行起来。以下是 LiteIDE 输出的控制台日志：

```
c:/go/bin/go.exe build -i [C:/project/go/k8s/src/github.com/GoogleCloudPlatform/
kubernetes/plugin/cmd/kube-scheduler]
成功：进程退出代码 0。
C:/project/go/k8s/src/github.com/GoogleCloudPlatform/kubernetes/plugin/cmd/
kube-scheduler/kube-scheduler.exe [C:/project/go/k8s/src/github.com/GoogleCloud
Platform/kubernetes/plugin/cmd/kube-scheduler]
W0717 16:05:26.742413 11344 server.go:83] Neither --kubeconfig nor --master was
specified. Using default API client. This might not work.
E0717 16:05:27.747413 11344 reflector.go:136] Failed to list *api.Node: Get
http://localhost:8080/api/v1/nodes?fieldSelector=spec.unschedulable=3Dfalse: dial
tcp 127.0.0.1:8080: ConnectEx tcp: No connection could be made because the target
machine actively refused it.
E0717 16:05:27.748413 11344 reflector.go:136] Failed to list *api.Pod: Get
http://localhost:8080/api/v1/pods?fieldSelector=spec.nodeName%21%3D: dial tcp
127.0.0.1:8080: ConnectEx tcp: No connection could be made because the target machine
actively refused it.
```

在 Kubernetes 的源码里包括不少单元测试，你可以在 LiteIDE 里运行通过，但有部分测试代码目前在 Windows 上无法通过，毕竟 Kubernetes 是为 Linux 打造的。接下来我们分析下 Kubernetes 源码的整体结构，Kubernetes 的源码总体分为 pkg、cmd、plugin、test 等顶级 package，其中 pkg 为 Kubernetes 的主体代码，cmd 为 Kubernetes 所有后台进程的代码（如 kube-apiserver 进程、kube-controller-manager 进程、kube-proxy 进程、kubelet 进程等），plugin 则包括一些插件及 kube-scheduler 的代码，test 包是 Kubernetes 的一些测试代码。

从总体来看，Kubernetes 1.0 的当前包结构还是有点乱，开源团队还在继续优化中，可以从源码的 TODO 注释中看出这一点。表 6.1 给出了 Kubernetes 当前主要 package 的源码分析结果。

表 6.1 Kubernetes 主要 package 的源码分析结果

package	模 块 用 途	类 数 量
admission	权限控制框架，采用了责任链模式、插件机制	少
api	Kubernetes 提供的 Rest API 接口的相关类，例如接口数据结构相关的 MetaData 结构、Volume 结构、Pod 结构、Service 结构等，以及数据格式验证转换工具类等，由于 API 是分版本的，所以这里是每个版本一个子 Package，例如 v1beta、v1 及 latest	中
apiserver	实现了 HTTP Rest 服务的一个基础性框架，用于 Kubernetes 的各种 Rest API 的实现，在 apiserver 包里也实现了 HTTP Proxy，用于转发请求（到其他组件，比如 Minion 节点上）	中
auth	3A 认证模块，包括用户认证、鉴权的相关组件	少

续表

package	模块用途	类数量
client	是Kubernetes中公用的客户端部分的相关代码，实现协议为HTTP Rest，用于提供一个具体的操作，例如对Pod、Service等的增删改查，这个模块也定义了kubeletClient，同时为了高效地进行对象查询，此模块也实现了一个带缓存功能的存储接口Store	多
cloudprovider	定义了云服务提供商运行Kubernetes所需的接口，包括TCP LoadBalancer的获取和创建；获取当前环境中的节点列表（节点是一个云主机）和节点的具体信息；获取Zone信息；获取和管理路由的接口等，默认实现了AWS、GCE、Mesos、OpenStack、RackSpace等云服务供应商的接口	中
controller	这部分提供了资源控制器的简单框架，用于处理资源的添加、变更、删除等事件的派发和执行，同时实现了Kubernetes的ReplicationController的具体逻辑	少
kubectl	Kubernetes的命令行工具kubectl的代码模块，包括创建Pod、服务、Pod扩容、Pod滚动升级等各种命令的具体实现代码	多
kubelet	Kubernetes的kubelet的代码模块，是Kubernetes的核心模块之一，定义了Pod容器的接口，提供了Docker与Rkt两种容器实现类，完成了容器及Pod的创建，以及容器状态的监控、销毁、垃圾回收等功能	多
master	Kubernetes的Master节点代码模块，创建NodeRegistry、PodRegistry、ServiceRegistry、EndpointRegistry等组件，并且启动Kubernetes自身的相关服务，服务的ClusterIP地址分配及服务的NodePort端口分配，也是在这里完成的	少
proxy	Kubernetes的服务代理和负载均衡相关功能的模块代码，目前实现了round-robin的负载均衡算法	少
registry	Kubernetes的NodeRegistry、PodRegistry、ReplicationControllerRegistry、ServiceRegistry、EndpointRegistry、PersistentVolumeRegistry等注册表服务的接口及对应Rest服务的相关代码	多
runtime	为了让多个API版本共存，需要采用一些设计来完成不同API版本的数据结构的转换，API中数据对象的Encode/Decode逻辑也最好集中化，Runtime包就是为了这个目的而设计的	少
volume	实现了Kubernetes的各种Volume类型，分别对应亚马逊ESB存储、谷歌GCE的存储、Linux Host目录存储、GlusterFS存储、iSCSI存储、NFS存储、RBD存储等，volume包同时实现了Kubernetes容器的Volume卷的挂载、卸载功能	多
cmd	包括了Kubernetes所有后台进程的代码（如kube-apiserver进程、kube-controller-manager进程、kube-proxy进程、kubelet进程等），而这些进程具体的业务逻辑代码则都在pkg中实现了	
plugin	子包cmd/kuber-scheduler实现了Schedule Server的框架，用于执行具体的Scheduler的调度，pkg/admission子包则实现了Admission权限框架的一些默认实现类，例如alwaysAdmit、alwaysDeny等；pkg/auth子包实现了权限认证框架(auth包的)里定义的认证接口类，例如HTTP BasicAuth、X509证书认证；pkg/scheduler子包则定义了一些具体的Pod调度器(Scheduler)	中

6.2 kube-apiserver 进程源码分析

Kubernetes API Server 是由 kube-apiserver 进程实现的，它运行在 Kubernetes 的管理节点——Master 上并对外提供 Kubernetes Restful API 服务，它提供的主要是与集群管理相关的 API 服务，例如校验 pod、service、replication controller 的配置并存储到后端的 etcd Server 上。下面我们分别对其启动过程、关键代码分析及设计总结等进行深入讲解。

6.2.1 进程启动过程

kube-apiserver 进程的入口类源码位置如下：

[github.com/GoogleCloudPlatform/kubernetes/cmd/kube-apiserver/apiserver.go](https://github.com/GoogleCloudPlatform/kubernetes/blob/master/cmd/kube-apiserver/apiserver.go)

入口 main() 函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    rand.Seed(time.Now().UTC().UnixNano())

    s := app.NewAPIServer()
    s.AddFlags(pflag.CommandLine)

    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()

    verflag.PrintAndExitIfRequested()

    if err := s.Run(pflag.CommandLine.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

上述代码的核心为下面三行，创建一个 APIServer 结构体并将命令行启动参数传入，最后启动监听：

```
s := app.NewAPIServer()
s.AddFlags(pflag.CommandLine)
s.Run(pflag.CommandLine.Args())
```

我们先来看看都有哪些常用的命令行参数被传递给了 APIServer 对象，下面是运行在 Master

节点的 `kube-apiserver` 进程的命令行信息：

```
/usr/bin/kube-apiserver --logtostderr=true --etcd_servers=http://127.0.0.1:4001 --address=0.0.0.0 --port=8080 --kubelet_port=10250 --allow_privileged=false --service-cluster-ip-range=10.254.0.0/16
```

可以看到关键的几个参数有 `etcd_servers` 的地址、`APIServer` 绑定和监听的本地地址、`kubelet` 的运行端口及 `Kubernetes` 服务的 `clusterIP` 地址。

下面是 `app.NewAPIServer()` 的代码，我们看到这里的控制还是很全面的，包括安全控制（`CertDirectory`、`HTTPS` 默认启动）、权限控制（`AuthorizationMode`、`AdmissionControl`）、服务限流控制（`APIRate`、`APIBurst`）等，这些逻辑说明了 `APIServer` 是按照企业级平台的标准所设计和实现的。

```
func NewAPIServer() *APIServer {
    s := APIServer{
        InsecurePort:          8080,
        InsecureBindAddress:   util.IP(net.ParseIP("127.0.0.1")),
        BindAddress:           util.IP(net.ParseIP("0.0.0.0")),
        SecurePort:            6443,
        APIRate:               10.0,
        APIBurst:              200,
        APIPrefix:             "/api",
        EventTTL:              1 * time.Hour,
        AuthorizationMode:     "AlwaysAllow",
        AdmissionControl:      "AlwaysAdmit",
        EtcdPathPrefix:         master.DefaultEtcdPathPrefix,
        EnableLogsSupport:     true,
        MasterServiceNamespace: api.NamespaceDefault,
        ClusterName:            "kubernetes",
        CertDirectory:          "/var/run/kubernetes",

        RuntimeConfig: make(util.ConfigurationMap),
        KubeletConfig: client.KubeletConfig{
            Port:      ports.KubeletPort,
            EnableHttps: true,
            HTTPTimeout: time.Duration(5) * time.Second,
        },
    }

    return &s
}
```

创建了 `APIServer` 结构体实例后，`apiserver.go` 将此实例传入子包 `app/server.go` 的 `func(*APIServer) Run([]string)` 方法里，最终绑定本地端口并创建一个 `HTTP Server` 与一个 `HTTPS Server`，从而完成整个进程的启动过程。

Run 方法的代码有很多，这里就不再列出源码，对该方法的源码解读如下。

- (1) 调用 verifyClusterIPFlags 方法，验证 ClusterIP 参数是否已设置及是否有效。
- (2) 验证 etcd-servers 的参数是否已设置。
- (3) 如果初始化 CloudProvider，且没有 CloudProvider 的参数，则日志告警并继续。
- (4) 根据 KubeletConfig 的配置参数，调用 pkg/Client/kubeclient.go 中的方法 NewKubeletClient() 创建一个 kubelet Client 对象，这其实是一个 HTTPKubeletClient 实例，目前只用于 kubelet 的健康检查 (KubeletHealthChecker)。
- (5) 判断哪些 API Version 需要关闭，目前在 1.0 代码中默认关闭了 v1beta3 的 API 版本。
- (6) 创建一个 Kubernetes 的 RestClient 对象，具体的代码在 pkg/client/helper.go 的 TransportFor() 方法里完成，通过它完成 Pod、Replication Controller 及 Kubernetes Service 等对象的 CRUD 操作。
- (7) 创建用于访问 etcd Server 的客户端，具体代码在 newEtcd() 方法里实现，从代码调用中可以看出，Kubernetes 采用的是 github.com/coreos/go-etcd/client.go 这个客户端实现。
- (8) 建立鉴权 (Authenticator)、授权 (Authorizer)、服务许可框架和插件 (AdmissionControl) 的相关代码逻辑。
- (9) 获取和设置 APIServer 的 ExternalHost 的名称，如果没有提供 ExternalHost 参数，且 Kubernetes 运行在谷歌的 GCE 云平台上，则尝试通过 CloudProvider 接口获取本机节点的外部 IP 地址。
- (10) 如果运行在云平台中，则安装本机的 SSH Key 到 Kubernetes 集群中的所有虚拟机上。
- (11) 用 APIServer 的数据及上述过程中创建的一些对象 (KubeletClient、etcdClient、authenticator、admissionController 等) 作为参数，构造 Kubernetes Master 的 Config 结构 (pkg/master/master.go)，以此生成一个 Master 实例，具体代码在 master.go 中的 New (c *Config) 方法里。
- (12) 用上述创建的 Master 实例，分别创建 HTTP Server 及安全的 HTTPS Server 来开始监听客户端的请求，至此整个进程启动完毕。

6.2.2 关键代码分析

在 6.2.1 节里对 kube-apiserver 进程的启动过程进行了详细分析，我们发现 Kubernetes API Service 的关键代码就隐藏在 pkg/master/master.go 里，APIServer 这个结构体只不过是一个参数传递通道而已，它的数据最终传给了 pkg/master/master.go 里的 Master 结构体，下面是它的完整定义：

```
// Master contains state for a Kubernetes cluster master/api server.
```

```

type Master struct {
    // "Inputs", Copied from Config
    serviceClusterIPRange *net.IPN
    serviceNodePortRange  util.PortRange
    cacheTimeout          time.Duration
    minRequestTimeout     time.Duration

    mux                  apiserver.Mux
    muxHelper           *apiserver.MuxHelper
    handlerContainer    *restful.Container
    rootWebService      *restful.WebService
    enableCoreControllers bool
    enableLogsSupport   bool
    enableUISupport     bool
    enableSwaggerSupport bool
    enableProfiling     bool
    apiPrefix           string
    corsAllowedOriginList util.StringList
    authenticator        authenticator.Request
    authorizer           authorizer.Authorizer
    admissionControl    admission.Interface
    masterCount          int
    v1beta3              bool
    v1                  bool
    requestContextMapper api.RequestContextMapper

    // External host is the name that should be used in external (public internet)
    URLs for this master
    externalHost string
    // clusterIP is the IP address of the master within the cluster.
    clusterIP       net.IP
    publicReadWritePort int
    serviceReadWriteIP  net.IP
    serviceReadWritePort int
    masterServices   *util.Runner

    // storage contains the RESTful endpoints exposed by this master
    storage map[string]rest.Storage
    // registries are internal client APIs for accessing the storage layer
    // TODO: define the internal typed interface in a way that clients can
    // also be replaced
    nodeRegistry      minion.Registry
    namespaceRegistry namespace.Registry
    serviceRegistry   service.Registry
    endpointRegistry endpoint.Registry
}

```

```

    serviceClusterIPAllocator service.RangeRegistry
serviceNodePortAllocator service.RangeRegistry
// "Outputs"
    Handler          http.Handler
InsecureHandler http.Handler

// Used for secure proxy
dialer          apiserver.ProxyDialerFunc
tunnels         *util.SSHTunnelList
tunnelsLock     sync.Mutex
installSSHKey  InstallSSHKey
lastSync        int64 // Seconds since Epoch
lastSyncMetric  prometheus.GaugeFunc
clock           util.Clock
}

```

在这段代码里，除了之前我们熟悉的那些变量，又多了几个陌生的重要变量，接下来我们逐一对其进行分析讲解。

首先是类型为 `apiserver.Mux`（来自文件 `pkg/apiserver/apiserver.go`）的 `mux` 变量，下面是对它的定义：

```

// mux is an object that can register http handlers.
type Mux interface {
    Handle(pattern string, handler http.Handler)
    HandleFunc(pattern string, handler func(http.ResponseWriter, *http.Request))
}

```

如果你熟悉 Socket 编程，特别使用过或者研究过 HTTP Rest 的一些框架，那么对于这个 `Mux` 接口就再熟悉不过了，它是一个 HTTP 的多分器（Multiplexer），其实它也是 Golang HTTP 基础包里的 `http.ServeMux` 的一个接口子集，用于派发（Dispatch）某个 `Request` 路径（这里用 `pattern` 变量表示）到对应的 `http.Handler` 进行处理。实际上在 `master.go` 代码中是生成一个 `http.ServeMux` 对象并赋值给 `apiserver.Mux` 变量，在代码中还有强制类型转换的语句。从上述分析来看，`apiserver.Mux` 的引入是设计的一个败笔，并没有增加什么价值，反而增加了理解代码的难度。此外，为了更好地实现 Rest 服务，Kubernetes 在这里引入了一个第三方的 REST 框架：github.com/emicklei/go-restful。

`go-restful` 在 GitHub 上有 36 个贡献者，采用了“路由”映射的设计思想，并且在 API 设计中使用了流行的 Fluent Style 风格，使用起来酣畅淋漓，也难怪 Kubernetes 选择了它。下面是 `go-restful` 的优良特性。

- ◎ Ruby on Rails 风格的 Rest 路由映射，例如`/people/{person_id}/groups/{group_id}`。
- ◎ 大大简化了 Rest API 的开发工作。

- ◎ 底层实现采用 Golang 的 HTTP 协议栈，几乎没有限制。
- ◎ 拥有完整的单元包代码，很容易开发一个可测试的 Rest API。
- ◎ Google AppEngine ready。

go-restful 框架中的核心对象如下。

- ◎ restful.Container：代表一个 HTTP Rest 服务器，包括一组 restful.WebService 对象和一个 http.ServeMux 对象，使用 RouteSelector 进行请求派发。
- ◎ restful.WebService：表示一个 Rest 服务，由多个 Rest 路由（restful.Route）组成，这一组 Rest 路由共享同一个 Root Path。
- ◎ restful.Route：表示一个 Rest 路由，Rest 路由主要由 Rest Path、HTTP Method、输入输出类型（HTML/JSON）及对应的回调函数 restful.RouteFunction 组成。
- ◎ restful.RouteFunction：一个用于处理具体的 REST 调用的函数接口定义，具体定义为 type RouteFunction func(*Request, *Response)。

Master 结构体里包含了对 restful.Container 与 restful.WebService 这两个 go-restful 核心对象的引用，在接下来的 Master 对象的构造方法中（对应代码为 master.go 的 func New(c *Config) *Master）被初始化。那么，问题又来了，Kubernetes 的这么一堆 Rest API 又是在哪里定义的，是如何被绑定到 restful.Route 里的呢？

要理解这个问题，我们要首先弄清楚 Master 结构体中的变量：

```
storage map[string]rest.Storage
```

storage 变量是一个 Map，Key 为 Rest API 的 path，Value 为 rest.Storage 接口，此接口是一个通用的符合 Restful 要求的资源存储服务接口，每个服务接口负责处理一类（Kind）Kubernetes Rest API 中的数据对象——资源数据，只有一个接口方法：New()，New()方法返回该 Storage 服务所能识别和管理的某种具体的资源数据的一个空实例。

```
type Storage interface {
    New() runtime.Object
}
```

在运行期间，Kubernetes API Runtime 运行时框架会把 New()方法返回的空对象的指针传入 Codec.DecodeInto([]byte, runtime.Object) 方法中，从而完成 HTTP Rest 请求中的 Byte 数组反序列化逻辑。Kubernetes API Server 中所有对外提供服务的 Restful 资源都实现了此接口，这些资源包括 pods、bindings、podTemplates、replicationControllers、services 等，完整的列表就在 master.go 的 func (m *Master) init(c *Config) 中，下面是相关代码片段（截取部分代码）。

```
m.storage = map[string]rest.Storage{
    "pods":           podStorage.Pod,
```

```

    "pods/status":      podStorage.Status,
    "pods/log":         podStorage.Log,
    "pods/exec":        podStorage.Exec,
    "pods/portforward": podStorage.PortForward,
    "pods/proxy":       podStorage.Proxy,
    "pods/binding":    podStorage.Binding,
    "bindings":         podStorage.Binding,

    "podTemplates": podTemplateStorage,

    "replicationControllers": controllerStorage,
    "services":           service.NewStorage(m.serviceRegistry,
m.nodeRegistry, m.endpointRegistry, serviceClusterIPAllocator, serviceNodePort
Allocator, c.ClusterName),
    "endpoints":          endpointsStorage,
    "minions":            nodeStorage,

```

看到上面这段代码，你在潜意识里已经明白，这其实就是似曾相识的 Kubernetes Rest API 列表，storage 这个 Map 的 Key 就是 Rest API 的访问路径，Value 却不是之前说好的 restful.Route。聪明的你一定想到了答案：必然存在一个“转换适配”的方法来实现上述转换！这段不难理解但源码超长的方法就在 pkg/apiserver/api_installer.go 的下述方法里：

```
func (a *APIInstaller) registerResourceHandlers(path string, storage rest.
Storage, ws *restful.WebService, proxyHandler http.Handler)
```

上述方法把一个 path 对应的 rest.Storage 转换成一系列的 restful.Route 并添加到指针 restful.WebService 中。这个函数的代码之所以很长，是因为有各种情况要考虑，比如 pods/portforward 这种路径要处理 child，还要判断每种的 Storage 资源类型所支持的操作类型；比如是否支持 create、delete、update 及是否支持 list、watch、patcher 操作等，对各种情况都考虑以后，这个函数的代码量已接近 500 行！估计 Kubernetes 这段代码的作者也不大好意思，于是外面封装了简单函数：func(a *APIInstaller)Install，内部循环调用 registerResourceHandlers，返回最终的 restful.WebService 对象，此方法的主要代码如下：

```

// Installs handlers for API resources.
func (a *APIInstaller) Install() (ws *restful.WebService, errors []error) {
    // Register the paths in a deterministic (sorted) order to get a deterministic
    swagger spec.
    paths := make([]string, len(a.group.Storage))
    var i int = 0
    for path := range a.group.Storage {
        paths[i] = path
        i++
    }
    sort.Strings(paths)
    for _, path := range paths {

```

```

        if err := a.registerResourceHandlers(path, a.group.Storage[path], ws,
proxyHandler); err != nil {
            errors = append(errors, err)
        }
    }
    return ws, errors
}

```

为了区分 API 的版本，在 `apiserver.go` 里定义了一个结构体：`APIGroupVersion`。以下是其代码：

```

type APIGroupVersion struct {
    Storage map[string]rest.Storage
    Root    string
    Version string
    // ServerVersion controls the Kubernetes APIVersion used for common objects
    // in the apiserver
    // schema like api.Status, api.DeleteOptions, and api.ListOptions. Other
    // implementors may
    // define a version "v1beta1" but want to use the Kubernetes "v1beta3" internal
    // objects. If
    // empty, defaults to Version.
    ServerVersion string

    Mapper meta.RESTMapper

    Codec    runtime.Codec
    Typer    runtime.ObjectTyper
    Creator  runtime.ObjectCreator
    Convertor runtime.ObjectConvertor
    Linker   runtime.SelfLinker

    Admit   admission.Interface
    Context  api.RequestContextMapper

    ProxyDialerFn    ProxyDialerFunc
    MinRequestTimeout time.Duration
}

```

我们注意到 `APIGroupVersion` 是与 `rest.Storage Map` 捆绑的，并且绑定了相应版本的 `Codec`、`Convertor` 用于版本转换，这样就很容易理解 Kubernetes 是怎样区分多版本 API 的 Rest 服务的。以下是过程详解。

首先，在 `APIGroupVersion` 的 `InstallREST(container *restful.Container)` 方法里，用 `Version` 变量来构造一个 Rest API Path 前缀并赋值给 `APIInstaller` 的 `prefix` 变量，并调用它的 `Install()` 方法完成 Rest API 的转换，代码如下：

```
func (g *APIGroupVersion) InstallREST(container *restful.Container) error {
```

```
    info := &APIRequestInfoResolver{util.NewStringSet(strings.TrimPrefix(g.Root,
"/")), g.Mapper}
    prefix := path.Join(g.Root, g.Version)
    installer := &APIInstaller{
        group:      g,
        info:       info,
        prefix:     prefix,
        minRequestTimeout: g.MinRequestTimeout,
        proxyDialerFn:   g.ProxyDialerFn,
    }
    ws, registrationErrors := installer.Install()
    container.Add(ws)
```

接着，在 APIInstaller 的 Install()方法里用 prefix (API 版本) 前缀生成 WebService 的相对根路径：

```
func (a *APIInstaller) newWebService() *restful.WebService {
    ws := new(restful.WebService)
    ws.Path(a.prefix)
    ws.Doc("API at"+ a.prefix +"version"+ a.group.Version)
    // TODO: change to restful.MIME_JSON when we set content type in client
    ws.Consumes("*/*")
    ws.Produces(restful.MIME_JSON)
    ws.ApiVersion(a.group.Version)

    return ws
}
```

最后，在 Kubernetes 的 Master 初始化方法 func (m *Master) init (c *Config) 里生成不同的 APIGroupVersion 对象，并调用 InstallRest() 方法，完成最终的多版本 API 的 Rest 服务装配流程：

```
if m.v1beta3 {
    if err := m.api_v1beta3().InstallREST(m.handlerContainer); err != nil {
        glog.Fatalf("Unable to setup API v1beta3: %v", err)
    }
    apiVersions = append(apiVersions, "v1beta3")
}
if m.v1 {
    if err := m.api_v1().InstallREST(m.handlerContainer); err != nil {
        glog.Fatalf("Unable to setup API v1: %v", err)
    }
    apiVersions = append(apiVersions, "v1")
}
```

至此，Rest API 的多版本问题还有最后一个需要澄清，即在不同的版本中接口的输入输出

参数的格式是有差别的，Kubernetes 是怎么处理这个问题的？

要弄明白这一点，我们首先要研究 Kubernetes API 里的数据对象的序列化、反序列化的实现机制。为了同时解决数据对象的序列化、反序列化与多版本数据对象的兼容和转换问题，Kubernetes 设计了一套复杂的机制，首先，它设计了 `conversion.Scheme` 这个结构体（`pkg/conversion/schema.go` 里），以下是对它的定义：

```
// Scheme defines an entire encoding and decoding scheme.
type Scheme struct {
    // versionMap allows one to figure out the go type of an object           //with
the given version and name.
    versionMap map[string]map[string]reflect.Type
    // typeToVersion allows one to figure out the version for a given //go object
The reflect.Type we index by should *not* be a pointer. If the same type
    // is registered for multiple versions, the last one wins.
    typeToVersion map[reflect.Type]string
    // typeToKind allows one to figure out the desired "kind" field //for a given
go object. Requirements and caveats are the same as typeToVersion.
    typeToKind map[reflect.Type][]string
    // converter stores all registered conversion functions. It also //has default
coverting behavior.
    converter *Converter
    // cloner stores all registered copy functions. It also has default
    // deep copy behavior.
    cloner *Cloner
    // Indent will cause the JSON output from Encode to be indented, iff it is true.
    Indent bool
    // InternalVersion is the default internal version. It is recommended that
    // you use "" for the internal version.
    InternalVersion string
    // MetaInsertionFactory is used to create an object to store and retrieve
    // the version and kind information for all objects. The default // uses
the keys "apiVersion" and "kind" respectively.
    MetaFactory MetaFactory
}
```

在上述代码中可以看到，`typeToVersion` 与 `versionMap` 属性是为了解决数据对象的序列化与反序列化问题，`converter` 属性则负责不同版本的数据对象转换问题，Kubernetes 这个设计思路简单方便地解决了多版本的序列化和数据转换问题，不得不赞！下面是 `conversion.Scheme` 里序列化、反序列化核心方法 `NewObject()` 的代码：通过查找 `versionMap` 里匹配的注册类型，以反射方式生成一个空的数据对象：

```
func (s *Scheme) NewObject(versionName, kind string) (interface{}, error) {
    if types, ok := s.versionMap[versionName]; ok {
        if t, ok := types[kind]; ok {
            return reflect.New(t).Interface(), nil
        }
    }
}
```

```

    }
    return nil, &notRegisteredErr{kind: kind, version: versionName}
}
return nil, &notRegisteredErr{kind: kind, version: versionName}
}

```

而 pkg/conversion/encode.go 与 decode.go 则在 conversion.Scheme 提供的基础功能之上，完成了最终的序列化、反序列化功能。下面是 encode.go 里的主方法 EncodeToVersion(..) 的关键代码片段：

```

//确定要转换的源对象的版本号和类别
objVersion, objKind, err := s.ObjectVersionAndKind(obj) 象
//生成目标版本的空对象
objOut, err := s.NewObject(destVersion, objKind)
//生成转换过程中所需的 Metadata 信息
flags, meta := s.generateConvertMeta(objVersion, destVersion, obj)
//调用 converter 的方法将源对象的数据填充到目标对象 objOut
err = s.converter.Convert(obj, objOut, flags, meta)
//用 JSON 将目标对象转换成 byte[] 数组，完成序列化过程
data, err = json.Marshal(obj)

```

再进一步，Kubernetes 在 conversion.Scheme 的基础上又做了一个封装工具类 runtime.Scheme，可以看作前者的代理类，主要增加了 fieldLabelConversionFuncs 这个 Map 属性，用于解决数据对象的属性名称的兼容性转换和校验，比如将需要兼容 Pod 的 spec.host 属性改为 spec.nodeName 的情况。

注意到 conversion.Scheme 只是实现了一个序列化与类型转换的框架 API，提供了注册资源数据类型与转换函数的功能，那么具体的资源数据对象类型、转换函数又是在哪个包里实现的呢？答案是 pkg/api。Kubernetes 为不同的 API 版本提供了独立的数据类型和相关的转换函数并按照版本号命名 Package，如 pkg/api/v1、pkg/api/v1beta3 等，而当前默认版本（内部版本）则存在于 pkg/api 目录下。

以 pkg/api/v1 为例，在每个目录里都包括如下关键源码：

- ◎ types.go 定义了 Rest API 接口里所涉及的所有数据类型，v1 版本有 2000 行代码；
- ◎ 在 conversion.go 与 conversion_generated.go 里定义了 conversion.Scheme 所需的从内部版本到 v1 版本的类型转换函数，其中 conversion_generated.go 中的代码有 5000 行之多，当然这是通过工具自动生成的代码；
- ◎ register.go 负责将 types.go 里定义的数据类型与 conversion.go 里定义的数据转换函数注册到 runtime.Schema 里。

pkg/api 里的 register.go 初始化生成并持有一个全局的 runtime.Scheme 对象，并将当前默认版本的数据类型（pkg/api/types.go）注册进去，相关代码如下：

```

var Scheme = runtime.NewScheme()
func init() {
    Scheme.AddKnownTypes("", 
        &Pod{},
        &PodList{},
        &PodStatusResult{},
        &PodTemplate{},
        &PodTemplateList{},
        &ReplicationControllerList{},
    //此次省略 30 多个数据类型
        &ServiceList{},
        &Service{},
        &NodeList{},
        &Node{},
    //省略
}

```

而 `pkg/api/v1/register.go` 与 `v1beta3` 下的 `register.go` 在初始化过程中分别把与版本相关的数据类型和转换函数注册到全局的 `runtime.Scheme` 中：

```

func init() {
    // Check if v1 is in the list of supported API versions.
    if !registered.IsRegisteredAPIVersion("v1") {
        return
    }

    // Register the API.
    addKnownTypes()
    addConversionFuncs()
    addDefaultingFuncs()
}

```

这样一来，其他地方都可以通过 `runtime.Scheme` 这个全局变量来完成 Kubernetes API 中的数据对象的序列化和反序列化逻辑了，比如 Kubernetes API Client 包就大量使用了它，下面是 `pkg/client/pods.go` 里 Pod 删除的 `Delete()` 方法的代码：

```

// Delete takes the name of the pod, and returns an error if one occurs
func (c *pods) Delete(name string, options *api.DeleteOptions) error {
    // TODO: to make this reusable in other client libraries
    if options == nil {
        return c.r.Delete().Namespace(c.ns).Resource("pods").Name(name).
Do().Error()
    }
    body, err := api.Scheme.EncodeToVersion(options, c.r.APIVersion())
    if err != nil {
        return err
    }
    return c.r.Delete().Namespace(c.ns).Resource("pods").Name(name).
Body(body).Do().Error()
}

```

```
}
```

清楚了 Kubernetes Rest API 中的数据对象的序列化机制及多版本的实现原理之后，我们接着分析下面这个重要流程的实现细节。

Kubernetes 中实现了 rest.Storage 接口的服务在转换成 restful.RouteFunction 以后，是怎样处理一个 Rest 请求并最终完成基于后端存储服务 etcd 上的具体操作过程的？

首先，Kubernetes 设计了一个名为“注册表”的 Package (pkg/registry)，这个 Package 按照 rest.Storage 服务所管理的资源数据的类型而划分为不同的子包，每个子包都由相同命名的一组 Golang 代码来完成具体的 Rest 接口的实现逻辑。

下面我们以 Pod 的 Rest 服务实现为例，其与“注册表”相关的代码位于 pkg/registry/pod 中，在 registry.go 里定义了 Pod 注册表服务的接口：

```
type Registry interface {
    // ListPods obtains a list of pods having labels which match selector.
    ListPods(ctx api.Context, label labels.Selector) (*api.PodList, error)
    // Watch for new/changed/deleted pods
    WatchPods(ctx api.Context, label labels.Selector, field fields.Selector,
resourceVersion string) (watch.Interface, error)
    // Get a specific pod
    GetPod(ctx api.Context, podID string) (*api.Pod, error)
    // Create a pod based on a specification.
    CreatePod(ctx api.Context, pod *api.Pod) error
    // Update an existing pod
    UpdatePod(ctx api.Context, pod *api.Pod) error
    // Delete an existing pod
    DeletePod(ctx api.Context, podID string) error
}
```

我们看到这个 Pod 注册表服务是针对 Pod 的 CRUD 的操作接口的一个定义，在入口参数中除了调用的上下文环境 api.Context，就是我们之前分析过的 pkg/api 包中的 Pod 这个资源数据对象。为了实现强类型的方法调用，在 registry.go 里定义了一个名为 storage 的结构体，storage 实现 Registry 接口，可以看作一种代理设计模式，因为具体的操作都是通过内部 rest.StandardStorage 来实现的。下面是截取的 registry.go 中的 create、update、delete 的源码：

```
func (s *storage) CreatePod(ctx api.Context, pod *api.Pod) error {
    _, err := s.Create(ctx, pod)
    return err
}

func (s *storage) UpdatePod(ctx api.Context, pod *api.Pod) error {
    _, _, err := s.Update(ctx, pod)
    return err
}
```

```

func (s *storage) DeletePod(ctx api.Context, podID string) error {
    _, err := s.Delete(ctx, podID, nil)
    return err
}

```

那么，这个实现了 rest.StandardStorage 通用接口的真正 Storage 又是什么？从 Master 对象的初始化函数中，我们发现了下面的相关代码：

```

func (m *Master) init(c *Config) {
    healthzChecks := []healthz.HealthzChecker{}
    m.clock = util.RealClock{}
    podStorage := podetcd.NewStorage(c.EtcdHelper, c.KubeletClient)
    podRegistry := pod.NewRegistry(podStorage.Pod)
}

```

Master 对象创建了一个私有变量 podStorage，其类型为 PodStorage（`pkg/registry/pod/etcd/etcd.go`），Pod 注册表服务实例（`podRegistry`）里真正的 Storage 是 `podStorage.Pod`。下面是 `podetcd` 的函数 `NewStorage` 中的关键代码：

```

func NewStorage(h tools.EtcdHelper, k client.ConnectionInfoGetter) PodStorage {
    store := &etcdgeneric.Etcd{
        NewFunc:     func() runtime.Object { return &api.Pod{} },
        NewListFunc: func() runtime.Object { return &api.PodList{} },
        .....
    }
    return PodStorage{
        Pod:          &REST{*store},
        Binding:      &BindingREST{store: store},
        Status:       &StatusREST{store: &statusStore},
        Log:          &LogREST{store: store, kubeletConn: k},
        Proxy:        &ProxyREST{store: store},
        Exec:         &ExecREST{store: store, kubeletConn: k},
        PortForward: &PortForwardREST{store: store, kubeletConn: k},
    }
}

```

在上述代码中我们看到：位于 `pkg/registry/generic/etcd/etcd.go` 里的 `etcd` 才是真正的 Storage 实现。而具体操作 `etcd` 的代码是靠 `tools.EtcdHelper` 这个类完成的，通过分析 `etcd.go` 里的 `func (*Etcd)Create(ctx api.Context, obj runtime.Object)` 方法，我们知道创建一个 `etcd` 里的键值对的关键逻辑如下。

- ◎ 获取对象的名字：`name, err := e.ObjectNameFunc(obj)`。
- ◎ 获取 Key：`key, err := e.KeyFunc(ctx, name)`。
- ◎ 生成一个空的 Object 对象：`out := e.NewFunc()`。
- ◎ 将键值对写入 etcd：在 `e.Helper.CreateObj(key, obj, out, ttl)` 方法中通过调用 `runtime.Codec` 完成从对象到字符串的转换，最终保存到 etcd 中。

◎ 回调创建完成后的处理逻辑：e.AfterCreate(out)。

注意到之前 PodStorage 创建 store 时重载了 ObjectNameFunc()、KeyFunc()、NewFunc() 等函数，于是完成了针对 Pod 的创建过程，Kubernetes API 服务中的其他数据对象也都遵循同样的设计模式。

进一步研究代码，我们发现 PodStorage 中的 Pod、Binding、Status 等属性是 pkg/api/rest/rest.go 中几个不同的 Rest 接口的实现，并且通过 etcdgeneric.Etcd 这个实例来完成 Pod 的一些具体操作，比如这里的 StatusREST。下面是其相关代码片段：

```
// StatusREST implements the REST endpoint for changing the status of a pod.
type StatusREST struct {
    store *etcdgeneric.Etcd
}
// New creates a new pod resource
func (r *StatusREST) New() runtime.Object {
    return &api.Pod{}
}
// Update alters the status subset of an object.
func (r *StatusREST) Update(ctx api.Context, obj runtime.Object) (runtime.Object, bool, error) {
    return r.store.Update(ctx, obj)
}
```

表 6.2 展现了 PodStorage 中的各个 XXXREST 接口与 pkg/api/rest/rest.go 里的相关 Rest 接口的一一对应关系。

表 6.2 PodStorage 中的各个 XXXREST 接口与 pkg/api/rest/rest.go 里的相关 Rest 接口的一一对应关系

PodStorage Rest 接口	对应 API Rest 框架的接口	接 口 功 能
REST	rest.Redirector rest.CreaterUpdater rest.Lister rest.Watcher rest.GracefulDeleter rest.Getter	重定向资源的路径 资源创建、更新接口 资源列表查询接口 Watcher 资源变化接口 支持延迟的资源删除接口 获取具体资源的信息接口
BindingREST	rest.Creater	创建资源的接口
StatusREST	Rest.Updater	更新资源的接口
LogREST	rest.GetterWithOptions	获取资源的接口
ExecREST\ProxyREST\ PortForwardREST	rest.Connector	连接资源的接口

其中 PodStorage.REST 接口究竟实现了哪些 API Rest 接口，这个比较隐晦，笔者也花费了一些时间来研究这个问题，这涉及 Go 语言的一个特殊特性：结构体内嵌一个其他类型的结构体指针，就可以使用内嵌结构体的方法，相当于面向对象语言中的“继承”。而 PodStorage.REST

恰恰嵌套了 `etcdgeneric.Etcd` 类型的匿名指针: `&REST{*store}`, 而 `etcdgeneric.Etcd` 则实现了 `rest.Creater`、`rest.Lister`、`rest.Watcher` 等资源管理接口的所有方法, `PodStorage.REST` 也“继承”了这些接口。

我们回头看看下面这段来自 `api_installer.go` 的 `registerResourceHandlers` 函数中的片段:

```
creater, isCreater := storage.(rest.Creater)
namedCreater, isNamedCreater := storage.(rest.NamedCreater)
lister, isLister := storage.(rest.Lister)
getter, isGetter := storage.(rest.Getter)
getterWithOptions, isGetterWithOptions := storage.(rest.GetterWithOptions)
deleter, isDeleter := storage.(rest.Deleter)
gracefulDeleter, isGracefulDeleter := storage.(rest.GracefulDeleter)
updater, isUpdater := storage.(rest.Updater)
patcher, isPatcher := storage.(rest.Patcher)
watcher, isWatcher := storage.(rest.Watcher)
_, isRedirector := storage.(rest.Redirector)
connecter, isConnecter := storage.(rest.Connecter)
storageMeta, isMetadata := storage.(rest.StorageMetadata)
```

上述代码对 `storage` 对象进行判断, 以确定并标记它所满足的 API Rest 接口类型, 而接下来的这段代码在此基础上确定此接口所包含的 `actions`, 后者则对应到某种 HTTP 请求方法 (`GET/POST/PUT/DELETE`) 或者 HTTP PROXY、WATCH、CONNECT 等动作:

```
actions = appendIf(actions, action{"GET", itemPath, nameParams, namer}, isGetter)
actions = appendIf(actions, action{"PATCH", itemPath, nameParams, namer}, isPatcher)
actions = appendIf(actions, action{"DELETE", itemPath, nameParams, namer}, isDeleter)
actions = appendIf(actions, action{"WATCH", "watch/" + itemPath, nameParams, namer}, isWatcher)
actions = appendIf(actions, action{"PROXY", "proxy/" + itemPath + "/{path:?}", proxyParams, namer}, isRedirector)
actions = appendIf(actions, action{"CONNECT", itemPath, nameParams, namer}, isConnecter)
```

我们注意到 `rest.Redirector` 类型的 `storage` 被当作 PROXY 进行处理, 由 `apiserver.ProxyHandler` 进行拦截, 并调用 `rest.Redirector` 的 `ResourceLocation` 方法获取到资源的处理路径 (可能包括一个非空的 `http.RoundTripper`, 用于处理执行 `Redirector` 返回的 URL 请求)。Kubernetes API Server 中 PROXY 请求存在的意义在于透明地访问其他某个节点 (比如某个 Minion) 上的 API。

最后, 我们来分析下 `registerResourceHandlers` 中完成从 `rest.Storage` 到 `restful.Route` 映射的最后一段关键代码。下面是 `rest.Getter` 接口的 `Storage` 的映射代码:

```
case "GET": // Get a resource.
```

```

var handler restful.RouteFunction
handler = GetResource(getter, reqScope)
doc := "read the specified " + kind
route := ws.GET(action.Path).To(handler).Filter(m).Doc(doc).
Param(ws.QueryParameter("pretty", "If 'true', then the output is pretty printed.
")).
Operation("read"+namespaced+kind+strings.Title(subresource)).
Produces(append(storageMeta.ProducesMIMETypes(action.Verb), "application/
json")...).
>Returns(http.StatusOK, "OK", versionedObject).Writes(versionedObject)

addParams(route, action.Params)
ws.Route(route)

```

上述代码首先通过函数 `GetResource()` 创建了一个 `restful.RouteFunction`，然后生成一个 `restful.route` 对象，最后注册到 `restful.WebService` 中，从而完成了 `rest.Storage` 到 Rest 服务的“最后一公里”通车。`GetResource()` 函数存在于 `pkg/apiserver/resthandler.go` 里，`resthandler.go` 提供了各种具体的 `restful.RouteFunction` 的实现函数，是真正触发 `rest.Storage` 调用的地方。下面是 `GetResource()` 方法的主要代码，可以看出这里是调用 `rest.Getter` 接口的 `Get()` 方法以返回某个资源对象：

```

func GetResource(r rest.Getter, scope RequestScope) restful.RouteFunction {
    return getResourceHandler(scope,
        func(ctx api.Context, name string, req *restful.Request) (runtime.Object,
error) {
            return r.Get(ctx, name)
        })
}

```

看了上面的代码，你可能会有一个疑问：“说好的权限控制呢？”别急，看看下面的资源创建的 `createHandler()` 代码：

```

if admit.Handles(admission.Create) {
    userInfo, _ := api.UserFrom(ctx)
    err = admit.Admit(admission.NewAttributesRecord(obj, scope.Kind,
namespace, name, scope.Resource, scope.Subresource, admission.Create, userInfo))
    if err != nil {
        errorJSON(err, scope.Codec, w)
        return
    }
}

```

资源的 `Update`、`Delete`、`Connect`、`Patch` 等操作都有类似的权限控制，从 `Admit` 的参数 `admission.Attributes` 的属性来看，第三方系统可以开发细粒度的权限控制插件，针对任意资源的任意属性进行细粒度的权限控制，因为资源对象本身都传递到参数中了。

对 Kubernetes Rest API Server 的复杂实现机制和调用流程的总结如下。

- ◎ 在 `pkg/api` 包里定义了 Rest API 中涉及的资源对象、提供的 Rest 接口、类型转换框架和具体转换函数、序列化反序列化等代码。其中，资源对象和转换函数按照版本分包，形成了 Kubernetes API Server 基础的框架，其中核心是各类资源（如 `Node`、`Pod`、`PodTemplate`、`Service` 等）及这些资源对应的 `rest.Storage`（Rest API 接口）。
- ◎ 在 `pkg/runtime` 包里最重要的对象是 `Schema`，它保存了 Kubernetes API Service 中注册的资源对象类型、转换函数等重要基础数据。另外，`runtime` 包也提供了获取 json/yaml 序列化、反序列化的 `Codec` 结构体，`runtime` 总体上与 `pkg/api` 密切关联，分离出来的目的是供其他模块方便使用。
- ◎ `pkg/registry` 包其实是把 `pkg/api` 中定义的各种资源对象所提供的 Rest 接口进一步规范定义并且实现对应的接口，其中 `generate/etcd/etcd.go` 里的 `etcd` 对象是一个真正实现了 `rest.Storage` 接口的基于 `etcd` 后端存储的服务框架，并且 Kubernetes 中的各种资源对象的具体 `Storage` 实现也是通过它来完成真正的“后端存储操作”。
- ◎ Kubernetes 采用了 `go-restful` 这个第三方的 Rest 框架，大大简化了 Rest 服务的开发，主要代码在 `pkg/apiserver` 源码包里。通过 `APIGroupVersion` 这个结构体可完成不同 API 版本的 Rest 路径映射，而 `api_installer.go` 则实现了从 Kubernetes `Rest.Storage` 接口到 `go-restful` 的映射连接逻辑，对应 `rest.Storage` 的具体 `restful.RouteFunction` 则在 `resthandler.go` 里实现。

6.2.3 设计总结

如果你耐心看完了上面的每一段文字和代码，而且尝试追踪源码来加深对 6.2.1 节内容的理解，那么笔者相信你对于 Kubernetes API Server 的设计的第一个评价就是：“太复杂、太反常了！不就是一个 Rest Server 么，如果用 Java 语言，我可以分分钟搞定一个！”当然，你肯定有以下或者更多的假设。

- ◎ 放弃多版本 API 的兼容需求。
- ◎ 只采用一个特定的后端存储实现。
- ◎ API 只接收一种输入输出格式，比如 JSON 或者 YAML，而不是两种或更多。
- ◎ 放弃 Watch 这种高难度的 API。
- ◎ 不实现 Proxy 代理。
- ◎ 不做可拔插的权限控制设计（或者根本没有）。

◎ 每新增一种资源类型，就从头写很多代码来实现该资源的 Rest 服务。

虽然代码很复杂，但我们不得不承认，Kubernetes API Server 是一个精心“设计”的系统。

什么样的设计是一个好的设计？这个问题没有标准答案，但有一点是大家都认可的：好的设计要尽量提供一种好的框架机制，方便未来增加新功能或者自定义扩展某些特性。我们以这个标准对 Kubernetes API Server 的设计进行评价，就会发现：它的设计真的很好。

我们先分析一下 Kubernetes API Server 的“领域模型”。API Server 里的 Rest 服务都是针对某个“资源对象”的操作，这些操作可以分为新增、修改、列表输出、删除、Watch 变化、代理请求及连接资源等基础操作，大多数操作都是与后端存储的交互。因为只是基本的资源数据对象的增、删、改、查，所以主体逻辑是通用的，比如序列化、反序列化、基于 Key-Value 的存储，以及这个过程中的数据校验和权限控制等问题。

通过以上分析，我们发现这个系统的核心对象只有两个：资源对象与操作资源对象的 Storage 服务。虽然各个资源的 Storage 服务的主体功能相同，都是将资源存储到 etcd 这个 Key-Value 后端存储系统上并提供相关操作，但不同类型资源的 Storage 服务的接口和具体逻辑还是有差别的，比如某类资源是不允许更新的，有些资源则允许“Connect”，所以这里的设计是 Kubernetes API Server 的最有代表性的经典设计——资源服务接口的细分与组合设计。

如图 6.2 所示是此设计的全景图(以 Pod 资源对象为例)。资源服务接口被拆分为 rest.Create、rest.Updater、rest.CreateUpdate（组合了 Create 与 Updater 接口）、rest.GracefulDelete（支持延迟删除资源的接口）、rest.Patcher（组合更新与 Get 接口）、rest.Connect（开启 HTTP 连接到该资源进行操作，比如连接到一个 Pod 中执行某个 bash 命令）等 10 个细分接口。

考虑到大多数资源对象都需要基本的 CRUD 接口，这就是 rest.StandardStorage 这个聚合型“标准存储服务”接口出现的原因。而作为 StandardStorage 的默认实现，pkg/registry/generic/etcd/etcd.go 里 etcd 这个对象实现了基于 etcd 后端存储的所有具体操作，而各种资源的 Storage 服务则通过将请求代理到 etcd 对象上来完成具体的功能。

这里有点让人难以理解的是 PodStorage 与它的属性 Pod 的关系，其实 PodStorage 这个对象是一个聚合了与 Pod 相关的各个资源的存储服务，多看一下它的定义就能立刻明白了：

```
// PodStorage includes storage for pods and all sub resources
type PodStorage struct {
    Pod          *REST
    Binding     *BindingREST
    Status      *StatusREST
    Log         *LogREST
    Proxy       *ProxyREST
    Exec        *ExecREST
    PortForward *PortForwardREST
}
```

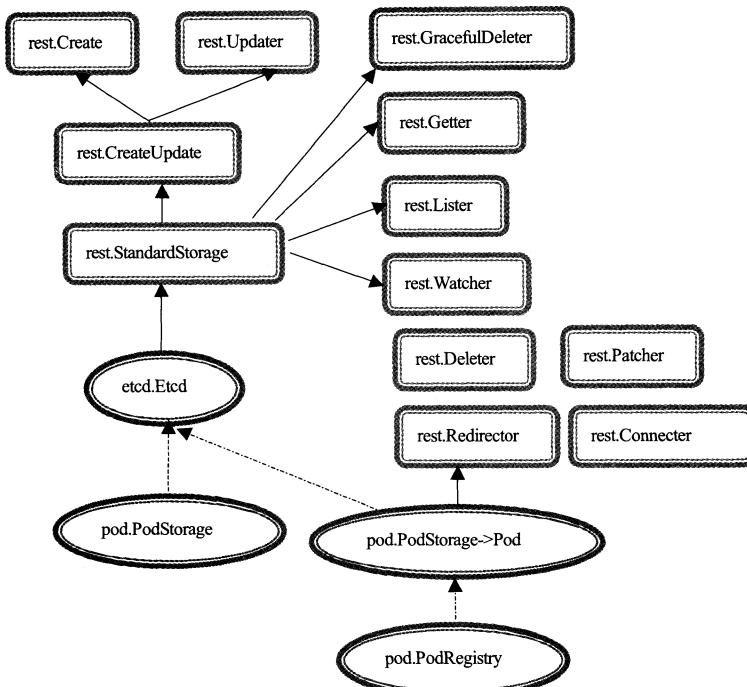


图 6.2 API Server 的 Storage 设计全景图

所以，这里的 PodStorage 应该重命名为 AllPodResStorage，而真正的 PodStorage 上就是里面的那个 Pod 变量，这个变量是对 etcd 实例的一个引用，然后又实现了 rest.Redirector 接口。现在你终于能理解 PodRegistry 引用 Pod 变量而不是 PodStorage 来实现 Pod 操作的真正原因了吧？

最后，我们来说说 PodRegistry 存在的目的。从之前的代码分析来看，一个来自外部的针对某个资源的 Rest API 发起的请求最后落到对应资源的 rest.Storage 对象上，由 restful.RouteFunction 调用此对象的相关方法完成资源的操作并生成应答返回给客户端，这个过程并没有涉及对应资源的 Registry 服务。那么问题来了，资源的 Registry 接口存在的理由是什么呢？答案很简单，对比 Storage 接口与 Registry 中的资源创建方法的签名，下面是二者的源码对比，后者更符合“手工调用”：

```

Storage 中创建通用的资源对象的接口
Create(ctx api.Context, obj runtime.Object) (runtime.Object, error)
PodRegistry 中创建 Pod 资源的接口
CreatePod(ctx api.Context, pod *api.Pod) error
  
```

在 Kubernetes API Server 中为每类资源都创建并提供了一个 Registry 接口服务的目的是供内部模块的编程使用，而非对外提供服务，很多文档都错误理解了这个问题。

本节最后给出了如图 6.3 所示的经典的 Kubernetes 的 Master 节点数据流图，此刻这个图在你眼里可能已经什么都不算了，因为你已经洞穿了幕后的一切。

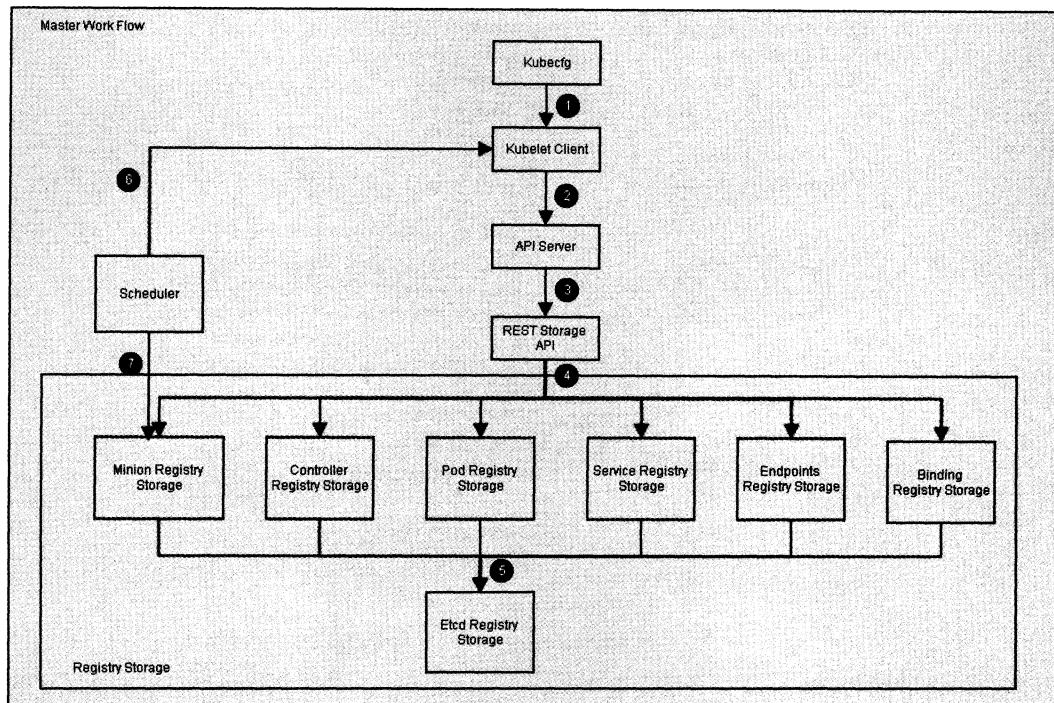


图 6.3 Master 节点数据流图

6.3 kube-controller-manager 进程源码分析

运行在 Master 节点上的第 2 个进程就是 `kube-controller-manager` 进程，即 Controller Manager Server，Kubernetes 的核心进程之一，其主要目的是实现 Kubernetes 集群的故障检测和恢复的自动化工作，比如内部组件 `EndpointController` 控制器负责 `Endpoints` 对象的创建和更新；`ReplicationManager` 根据注册表中的 `ReplicationController` 的定义，完成 Pod 的复制或者移除，以确保复制数量的一致性；`NodeController` 负责 Minion 节点的发现、管理和监控。

6.3.1 进程启动过程

`kube-controller-manager` 进程的入口源码位置如下：

[github.com/GoogleCloudPlatform/kubernetes/cmd/kube-controller-manager/controller-manager.go](https://github.com/GoogleCloudPlatform/kubernetes/blob/master/cmd/kube-controller-manager/controller-manager.go)

入口 main() 函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewCMServer()
    s.AddFlags(pflag.CommandLine)
    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()
    verflag.PrintAndExitIfRequested()
    if err := s.Run(pflag.CommandLine.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

从源码可以看出，关键代码只有两行，创建一个 CMSServer 并调用 Run 方法启动服务。下面我们分析 CMSServer 这个结构体，它是 Controller Manager Server 进程的主要上下文数据结构，存放一些关键参数，表 6.3 是对 CMSServer 里的关键参数的解释。

表 6.3 CMSServer 的重要属性

属性名	默认	含义
ConcurrentEndpointSyncs	5 秒	并发执行的 Endpoint 的同步任务的数量
ConcurrentRCSyncs	5 秒	并发执行的 Replication Controller 的同步任务的数量
NodeSyncPeriod	5 秒	从 CloudProvider 处同步 Node 节点的周期
NodeMonitorPeriod	5 秒	Node 节点监控的周期
ResourceQuotaSyncPeriod	10 秒	对资源的配额使用情况进行同步的周期
NamespaceSyncPeriod	5 分钟	Namespace 同步的周期
PVClaimBinderSyncPeriod	10 秒	对 PV（持久存储）和 PV 的申请进行同步的周期
PodEvictionTimeout	5 分钟	在 Node 失败的情况下，其上的 Pod 多久后才被删除
master		Kubernetes API Server 的访问地址

从上述这些变量来看，Controller Manager Server 其实就是一个“超级调度中心”，它负责定期同步 Node 节点状态、资源使用配额信息、Replication Controller、Namespace、Pod 的 PV 绑定等信息，也包括执行诸如监控 Node 节点状态、清除失败的 Pod 容器记录等一系列定时任务。

在 controller-manager.go 里创建 CMSServer 实例并把参数从命令行中传递到 CMSServer 后，就调用它的 func (s *CMSServer) Run (_ []string) 方法进入关键流程，这里首先创建一个 Rest Client 对象用于访问 Kubernetes API Server 提供的 API 服务：

```

        kubeClient, err := client.New(kubeconfig)
if err != nil {
    glog.Fatalf("Invalid API configuration: %v", err)
}

```

随后，创建一个 HTTP Server 以提供必要的性能分析（Performance Profile）和性能指标度量（Metrics）的 Rest 服务：

```

go func() {
    mux := http.NewServeMux()
    healthz.InstallHandler(mux)
    if s.EnableProfiling {
        mux.HandleFunc("/debug/pprof/", pprof.Index)
        mux.HandleFunc("/debug/pprof/profile", pprof.Profile)
        mux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
    }
    mux.Handle("/metrics", prometheus.Handler())

    server := &http.Server{
        Addr:    net.JoinHostPort(s.Address.String(),
strconv.Itoa(s.Port)),
        Handler: mux,
    }
    glog.Fatal(server.ListenAndServe())
}()

```

我们注意到性能分析的 Rest 路径是以/debug 开头的，表明是为了程序调试所用，事实上的确如此，这里的几个 Profile 选项都是针对当前 Go 进程的 Profile 数据，比如我们在 Master 节点上执行 curl 命令（地址为 `http://127.0.0.1:10252/debug/pprof/heap`）可以获取进程的当前堆栈信息，会输出如下信息：

```

heap profile: 4: 78112 [1109: 824584] @ heap/1048576
  1: 32768 [1: 32768] @ 0x402612 0x75ab95 0x771419 0x771379 0x565f08 0x46133f
0x400d10 0x4155a3 0x43e711
  1: 32768 [1: 32768] @ 0x408806 0x407968 0x97e591 0x9895aa 0x76099b 0xa2f400
0xa4e887 0x765dc4 0x557fbc 0x782fac 0x5fe5db 0x602ca7 0x462c92 0x400f06 0x415594
0x43e711
  1: 12288 [1: 12288] @ 0x4199fc 0x7df75d 0x5b585c 0x5b4947 0x5b405a 0x5aa472
0x5aa2b7 0x5aa188 0x5ad0d3 0x46291e 0x43e711
  1: 288 [1: 288] @ 0x415d6a 0x43276f 0x43510f 0x42fd37 0x4311f9 0x430ef5 0x43c136

```

其他还有 GC 回收、Symbol 查看、进程 30 秒内的 CPU 利用率、协程的阻塞状态等 Profile 功能，输出的数据格式符合 `google-perftools` 这个工具的要求，因此可以做运行期的可视化 Profile，以便排查当前进程潜在的问题或性能瓶颈。

性能指标度量目前主要收集和统计 Kubernetes API Server 的 Rest API 的调用情况，执行 curl

(<http://127.0.0.1:10252/metrics>)，可以看到输出中包括大量类似下面的内容：

```
rest_client_request_latency_microseconds{url="http://centos-master:8080/api/v1/namespaces/default/endpoints/%3Cname%3E",verb="GET",quantile="0.5"} 1448
rest_client_request_latency_microseconds{url="http://centos-master:8080/api/v1/namespaces/default/endpoints/%3Cname%3E",verb="GET",quantile="0.9"} 1699
rest_client_request_latency_microseconds{url="http://centos-master:8080/api/v1/namespaces/default/endpoints/%3Cname%3E",verb="GET",quantile="0.99"} 2093
```

这些指标有助于协助发现 Controller Manager Server 在调度方面的性能瓶颈，因此可以理解为什么会被包括到进程代码中去。

接下来，启动流程进入到关键代码部分。在这里，启动进程分别创建如下控制器，这些控制器的主要目的是实现资源在 Kubernetes API Server 的注册表中的周期性同步工作：

- ◎ EndpointController 负责对注册表中的 Kubernetes Service 的 Endpoints 信息的同步工作；
- ◎ ReplicationManager 根据注册表中对 ReplicationController 的定义，完成 Pod 的复制或者移除，以确保复制数量的一致性；
- ◎ NodeController 则通过 CloudProvider 的接口完成 Node 实例的同步工作；
- ◎ servicecontroller 通过 CloudProvider 的接口完成云平台中的服务的同步工作，这些服务目前主要是外部的负载均衡服务；
- ◎ ResourceQuotaManager 负责资源配额使用情况的同步工作；
- ◎ NamespaceManager 负责 Namespace 的同步工作；
- ◎ PersistentVolumeClaimBinder 与 PersistentVolumeRecycler 分别完成 PersistentVolume 的绑定和回收工作；
- ◎ TokensController、ServiceAccountsController 分别完成 Kubernetes 服务的 Token、Account 的同步工作。

创建并启动完成上述的控制器以后，各个控制器就开始独立工作，Controller Manager Server 启动完毕。

6.3.2 关键代码分析

在 6.3.1 节对 kube-controller-manager 进程的启动过程进行了详细分析，我们发现这个进程的主要逻辑就是启动一系列的“控制器”。这里以 Kubernetes 里比较关键的 Pod 副本（Pod Replica）数量的控制实现过程为例，来分析完成这个任务的“控制器”——ReplicationManager 具体是如何工作的。

首先，我们来看看 `ReplicationManager` 结构体的定义：

```
type ReplicationManager struct {
    kubeClient client.Interface
    podControl PodControlInterface

    // An rc is temporarily suspended after creating/deleting these many replicas.
    // It resumes normal action after observing the watch events for them.
    burstReplicas int
    // To allow injection of syncReplicationController for testing.
    syncHandler func(rcKey string) error

    // podStoreSynced returns true if the pod store has been synced at least once.
    // Added as a member to the struct to allow injection for testing.
    podStoreSynced func() bool

    // A TTLCache of pod creates/deletes each rc expects to see
    expectations RCEExpectationsManager
    // A store of controllers, populated by the rcController
    controllerStore cache.StoreToControllerLister
    // A store of pods, populated by the podController
    podStore cache.StoreToPodLister
    // Watches changes to all replication controllers
    rcController *framework.Controller
    // Watches changes to all pods
    podController *framework.Controller
    // Controllers that need to be updated
    queue *workqueue.Type
}
```

在上述结构体里，比较关键的几个属性如下。

- ◎ `kubeClient`: 用来访问 Kubernetes API Server 的 Rest 客户端，这里用来访问注册表中定义的 `ReplicationController` 对象并操作 Pod。
- ◎ `podControl`: 实现了 Pod 副本创建的函数，其实现类为 `RealPodControl`（位于 `kubernetes/pkg/controller/controller_utils.go`）。
- ◎ `syncHandler`: 是 RC (`ReplicationController`) 的同步实现方法，完成具体的 RC 同步逻辑（创建 Pod 副本时调用 `PodControl` 的相关方法），在代码中其被赋值为 `ReplicationManager.syncReplicationController` 方法。
- ◎ `expectations`: 是 Pod 副本在创建、删除过程中的流控机制的重要组成部分。
- ◎ `controllerStore`: 是一个具备本地缓存功能的通用的资源存储服务，这里存放 `framework.Controller` 运行过程中从 Kubernetes API Server 同步过来的资源数据，目的是减轻资源同步过程中对 Kubernetes API Server 造成的访问压力并提高资源同步的效率。

- ◎ `rcController`: `framework.Controller` 的一个实例，用来实现 RC 同步的任务调度逻辑。
- ◎ `framework.Controller`: 是 `kube-controller-manager` 里设计的用于资源对象同步逻辑的专用任务调度框架。
- ◎ `podStore`: 类似于 `controllerStore` 的作用，用来存取和获取 Pod 资源对象。
- ◎ `podController`: 类似于 `rcController` 的作用，用来实现 Pod 同步的任务调度逻辑。

理解了 `ReplicationManager` 结构体的重要参数及其作用之后，我们来看 `controller.NewReplicationManager(kubeClient client.Interface, burstReplicas int) *ReplicationManager` 这个构造函数中的关键代码，注意到这里通过调用 `framework.NewInformer()` 方法先后创建了用于 RC 同步及 Pod 同步的 `framework.Controller`。下面是 `framework.NewInformer()` 方法的源码：

```
func NewInformer(
    lw cache.ListerWatcher,
    objType runtime.Object,
    resyncPeriod time.Duration,
    h ResourceEventHandler,
) (cache.Store, *Controller) {
    clientState := cache.NewStore(DeletionHandlingMetaNamespaceKeyFunc)
    fifo := cache.NewDeltaFIFO(cache.MetaNamespaceKeyFunc, nil, clientState)
    cfg := &Config{
        Queue:           fifo,
        ListerWatcher:   lw,
        ObjectType:      objType,
        FullResyncPeriod: resyncPeriod,
        RetryOnError:    false,
        Process: func(obj interface{}) error {
            // from oldest to newest
            for _, d := range obj.(cache.Deltas) {
                switch d.Type {
                case cache.Sync, cache.Added, cache.Updated:
                    if old, exists, err := clientState.Get(d.Object); err == nil
&& exists {
                        if err := clientState.Update(d.Object); err != nil {
                            return err
                        }
                        h.OnUpdate(old, d.Object)
                    } else {
                        if err := clientState.Add(d.Object); err != nil {
                            return err
                        }
                        h.OnAdd(d.Object)
                    }
                case cache.Deleted:
                    if err := clientState.Delete(d.Object); err != nil {

```

```

        return err
    }
    h.OnDelete(d.Object)
}
}
return nil
},
}
return clientState, New(cfg)
}

```

在上述代码中，`lw(ListerWatcher)`用来获取和监测资源对象的变化，而 `fifo` 则是一个 DeltaFIFO 的 Queue，用来存放变化的资源（需要同步的资源）。当 Controller 框架发现有变化的资源需要处理时，就会将新资源与本地缓存 `clientState` 中的资源进行对比，然后调用相应的资源处理函数 `ResourceEventHandler` 的方法，完成具体的处理逻辑。下面是针对 RC 的 `ResourceEventHandler` 的具体实现：

```

framework.ResourceEventHandlerFuncs{
    AddFunc: rm.enqueueController,
    UpdateFunc: func(old, cur interface{}) {
        oldRC := old.(*api.ReplicationController)
        curRC := cur.(*api.ReplicationController)
        if oldRC.Status.Replicas != curRC.Status.Replicas {
            glog.V(4).Infof("Observed updated replica count for rc: %v,
%d->%d", curRC.Name, oldRC.Status.Replicas, curRC.Status.Replicas)
        }
        rm.enqueueController(cur)
    },
    DeleteFunc: rm.enqueueController,
}

```

在上述源码中，我们看到当 RC 里 Pod 的副本数量属性发生变化以后，`ResourceEventHandler` 就将此 RC 放入 `ReplicationManager` 的 `queue` 队列中等待处理，为什么没有在这个 `handler` 函数中直接处理而是先放入队列再异步处理呢？最主要的一个原因是 Pod 副本创建的过程比较耗时。Controller 框架把需要同步的 RC 对象放入 `queue` 以后，接下来是谁在“消费”这个队列呢？答案就在 `ReplicationManager` 的 `Run()` 方法中：

```

func (rm *ReplicationManager) Run(workers int, stopCh <-chan struct{}) {
    defer util.HandleCrash()
    go rm.rcController.Run(stopCh)
    go rm.podController.Run(stopCh)
    for i := 0; i < workers; i++ {
        go util.Until(rm.worker, time.Second, stopCh)
    }
    <-stopCh
    glog.Infof("Shutting down RC Manager")
}

```

```
    rm.queue.ShutDown()
}
```

上述代码首先启动 rcController 与 podController 这两个 Controller，启动之后，这两个 Controller 就分别开始拉取 RC 与 Pod 的变动信息，随后又启动 N 个协程并发处理 RC 的队列，其中 func Until(f func(), period time.Duration, stopCh <-chan struct{}) 方法的逻辑是按照指定的周期 period 执行方法 f。下面是 ReplicationManager 的 worker 方法的源码，负责从 RC 队列中拉取 RC 并调用 rm 的 syncHandler 方法完成具体处理：

```
func (rm *ReplicationManager) worker() {
    for {
        func() {
            key, quit := rm.queue.Get()
            if quit {
                return
            }
            defer rm.queue.Done(key)
            err := rm.syncHandler(key.(string))
            if err != nil {
                glog.Errorf("Error syncing replication controller: %v", err)
            }
        }()
    }
}
```

从 ReplicationManager 的构造函数中我们得知：syncHandler 在这里其实是 func (rm *ReplicationManager) syncReplicationController(key string) 方法。下面是该方法的源码：

```
func (rm *ReplicationManager) syncReplicationController(key string) error {
    startTime := time.Now()
    defer func() {
        glog.V(4).Infof("Finished syncing controller %q (%v)", key, time.
Now().Sub(startTime))
    }()

    obj, exists, err := rm.controllerStore.Store.GetByKey(key)
    if !exists {
        glog.Infof("Replication Controller has been deleted %v", key)
        rm.expectations.DeleteExpectations(key)
        return nil
    }
    if err != nil {
        glog.Infof("Unable to retrieve rc %v from store: %v", key, err)
        rm.queue.Add(key)
        return err
    }
    controller := *obj.(*api.ReplicationController)
```

```

        if !rm.podStoreSynced() {
            // Sleep so we give the pod reflector goroutine a chance to run.
            time.Sleep(PodStoreSyncedPollPeriod)
            glog.Infof("Waiting for pods controller to sync, requeueing rc %v",
controller.Name)
            rm.enqueueController(&controller)
            return nil
        }

        rcNeedsSync := rm.expectations.SatisfiedExpectations(&controller)
        podList, err := rm.podStore.Pods(controller.Namespace).List(labels.Set
(controller.Spec.Selector).AsSelector())
        if err != nil {
            glog.Errorf("Error getting pods for rc %q: %v", key, err)
            rm.queue.Add(key)
            return err
        }

        filteredPods := filterActivePods(podList.Items)
        if rcNeedsSync {
            rm.manageReplicas(filteredPods, &controller)
        }

        if err := updateReplicaCount(rm.kubeClient.ReplicationControllers(controller.
Namespace), controller, len(filteredPods)); err != nil {
            rm.enqueueController(&controller)
        }
        return nil
    }
}

```

在上述代码里有一个重要的流控变量 `rcNeedsSync`。为了限流，在 RC 同步逻辑的过程中，一个 RC 每次最多执行 N 个 Pod 的创建、删除，如果某个 RC 的同步过程涉及的 Pod 副本数量超过 `burstReplicas` 这个阈值，就会采用 `RCExpectations` 机制进行限流。`RCExpectations` 对象可以理解为一个简单的规则：即在限定的时间内执行 N 次操作，每次操作都使计数器减一，计数器为零表示 N 个操作已经完成，可以进行下一批次的操作了。

Kubernetes 为什么会设计这样一个流程控制机制？其实答案很简单——为了公平。因为谷歌的开发 Kubernetes 的资深大牛们早已预见到某个 RC 的 Pod 副本一次扩容至 100 倍的极端情况可能真实发生，如果没有流控机制，则这个巨无霸的 RC 同步操作会导致其他众多“散户”崩溃！这绝对不是谷歌的理念。

接着看上述代码里所调用的 `ReplicationManager` 的 `manageReplicas` 方法，这是 RC 同步的具体逻辑实现，此方法采用了并发调用的方式执行批量的 Pod 副本操作任务，相关代码如下：

```
wait := sync.WaitGroup{}
```

```

        wait.Add(diff)
       	glog.V(2).Infof("Too few %q/%q replicas, need %d, creating %d",
controller.Namespace, controller.Name, controller.Spec.Replicas, diff)
       	for i := 0; i < diff; i++ {
           	go func() {
               	defer wait.Done()
               	if err := rm.podControl.createReplica(controller.Namespace,
controller); err != nil {
                   	glog.V(2).Infof("Failed creation, decrementing expectations for
controller %q/%q", controller.Namespace, controller.Name)
                   	rm.expectations.CreationObserved(controller)
                   	util.HandleError(err)
               	}
            }()
        }
    wait.Wait()
}

```

追踪至此，我们才看到创建 Pod 副本的真正代码在 PodControl.createReplica()方法里，而此方法的具体实现方法则是 RealPodControl.createReplica()，位于 controller_utils.go 里。通过分析该方法，我们可以知道创建 Pod 副本的过程就是创建一个 Pod 资源对象，并把 RC 中定义的 Pod 模板赋值给该 Pod 对象，并且 Pod 的名字用 RC 的名字做前缀，最后调用 Kubernetes Client 将 Pod 对象通过 Kubernetes API Server 写入后端的 etcd 存储中。

在本节最后，我们来分析一下 Controller 框架中如何实现资源对象的查询和监听逻辑并且在资源发生变动时回调 Controller.Config 对象中的 Process 方法：func(obj interface{})，最终完成整个 Controller 框架的闭环过程。

首先，在 Controller 框架中构建了 Reflector 对象以实现资源对象的查询和监听逻辑，它的源码位于 pkg/client/cache/reflector.go 中，我们看一下这个对象的数据结构就基本明白了其工作原理：

```

// Reflector watches a specified resource and causes all changes to be reflected
in the given store.
type Reflector struct {
    // The type of object we expect to place in the store.
    expectedType reflect.Type
    // The destination to sync up with the watch source
    store Store
    // listerWatcher is used to perform lists and watches.
    listerWatcher ListerWatcher
    // period controls timing between one watch ending and
    // the beginning of the next one.
    period      time.Duration
    resyncPeriod time.Duration
    // lastSyncResourceVersion is the resource version token last
}

```

```

    // observed when doing a sync with the underlying store
    // it is thread safe, but not synchronized with the underlying store
    lastSyncResourceVersion string
    // lastSyncResourceVersionMutex guards read/write access to
lastSyncResourceVersion
    lastSyncResourceVersionMutex sync.RWMutex
}

```

核心思路就是通过 `listerWatcher` 去获取资源列表并监听资源的变化，然后存储到 `store` 中。这里你可能有个疑问，这个 `store` 究竟是哪个对象？是 `ReplicationManager` 里的 `controllerStore` 还是 `framework.NewInformer()` 方法里创建的 `fifo` 队列？

下面的两段来自 `pkg/controller/framework/controller.go` 的代码会告诉我们答案。

首先是来自 `Controller` 的 `run` 方法 `func (c *Controller) Run(stopCh <-chan struct{})` 的代码片段：

```

r := cache.NewReflector(
    c.config.ListerWatcher,
    c.config.ObjectType,
    c.config.Queue,
    c.config.FullResyncPeriod,
)

```

然后是来自 `Controller` 的 `NewInformer` 方法 `func NewInformer(lw cache.ListerWatcher, objType runtime.Object, resyncPeriod time.Duration, h ResourceEventHandler,) (cache.Store, *Controller)` 中的代码片段：

```

cfg := &Config{
    Queue:         fifo,
    ListerWatcher: lw,
    ObjectType:   objType,
    FullResyncPeriod: resyncPeriod,
    RetryOnError:  false,
}

```

分析上述代码，我们发现 `Reflector` 中的 `store` 其实是引用 `Controller.Config` 里的 `Queue` 属性，即 `fifo` 队列，而非 `ReplicationManager` 里的 `controllerStore`。我们费了这么大的劲，才弄明白这个问题，这告诉我们一个事实：编程中有良好的命名规则很重要。

下面这段代码是 `Controller` 从队列 `Queue` 中拉取资源对象并且交给 `Controller.Config` 对象中的 `Process` 方法 `func(obj interface{})` 进行处理，从而最终完成了整个 `Controller` 框架的闭环过程。

```

func (c *Controller) processLoop() {
    for {
        obj := c.config.Queue.Pop()
    }
}

```

```

        err := c.config.Process(obj)
        if err != nil {
            if c.config.RetryOnError {
                // This is the safe way to re-enqueue.
                c.config.Queue.AddIfNotPresent(obj)
            }
        }
    }
}

```

至于上述过程的调用则是在 Controller 启动（Run 方法）的最后一步里，Controller 框架定时每秒调用一次上述函数，代码如下：

```
util.Until(c.processLoop, time.Second, stopCh)
```

最后，给读者留一个源码解读的问题，即 ReplicationManager 里除了 RC Controller，又构造了一个用于 Pod 的 Controller，它的逻辑具体是怎样实现的？它与 RC Controller 是怎样交互的？

6.3.3 设计总结

相对于之前的 Kubernetes API Server 设计来说，Kubernetes Controller Server 的设计没有那么复杂，而且精彩依旧。不愧是大师的作品，Controller Framework 精巧细致的设计使得整个进程中各种资源对象的同步逻辑在代码实现方面保持了高度一致性与简捷性。此外，在关键资源 RC（Replication Controller）的同步逻辑中所采用的流控机制也简单、高效。

本节我们针对 Kubernetes Controller Server 中的精华部分——Controller Framework 的设计做一个整理分析。首先，framework.Controller 内部维护一个 Config 对象，保留了一个标准的消息、事件分发系统的三要素。

- ◎ 生产者：cache.ListerWatch。
- ◎ 队列：cache.cacheStore(Queue)。
- ◎ 消费者：用回调函数来模拟(framework.ResourceEventHandlerFuncs)。

由于生产者的逻辑比较复杂，在这个系统中也有其特殊性，即拉取资源并监控资源的变化，由此产生了真正的待处理任务，所以又设计了一个 ListerWatcher 接口，将底层的复杂逻辑“框架化”，放入 cache.Reflector 中，使用者只要简单地实现 ListerWatcher 接口的 ListFunc 与 WatchFunc 即可。另外，cache.Reflector 也是独立于 Controller Framework 的一个组件，隶属于 cache 包，它的功能是将任意资源对象拉取到本地缓存中并监控资源的变化，保持本地缓存的同步，其目标是减轻对 Kubernetes API Server 的请求压力。

图 6.4 给出了 Controller Framework 的整体架构设计图。

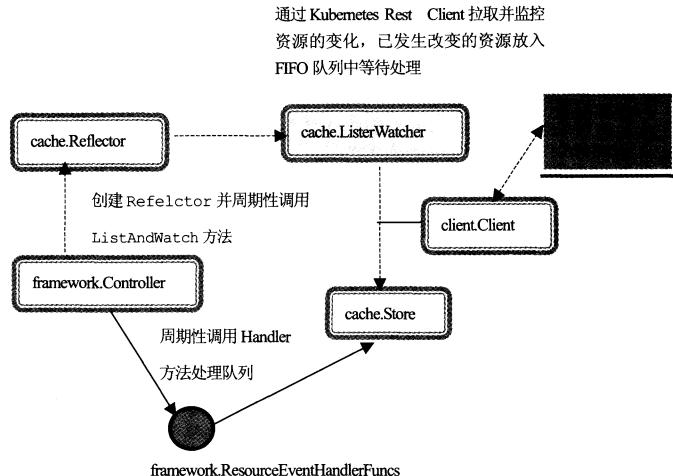


图 6.4 Controller Framework 的整体架构设计图

Kubernetes Controller Server 中所有涉及同步的资源都采用了 Controller Framework 框架来进行驱动，图 6.5 给出了整体设计示意图。

从图 6.5 可以看出，除了 Node、Route、Cloud Service 这三个资源依赖于 Kubernetes 所处的云计算环境，只能通过 CloudProvider 接口所提供的 API 来完成资源同步，其他资源都采用了 Controller Framework 框架来进行资源同步。图中的虚线箭头表示针对目标资源创建了一个 framework.Controller 对象，其中的某些资源如 RC、PV、Tokens 的同步过程需要获取并监听其他与之相关联的资源对象。这里只有 ResourceQuota 资源比较另类，它没有采用 Controller Framework，一个原因是 ResourceQuota 涉及很多资源对象，不大好应用 framework.Controller，另外一个原因可能是写 ResourceQuotaManager 的大牛拥有比较浪漫的情怀，看看下面这段 Kubernetes 中最优秀的代码吧：

```
func (rm *ResourceQuotaManager) Run(period time.Duration) {
    rm.syncTime = time.Tick(period)
    go util.Forever(func() { rm.synchronize() }, period)
}
```

核心代码翻译过来就是这个意思：从此他们过上了幸福的生活，一去不复返了！

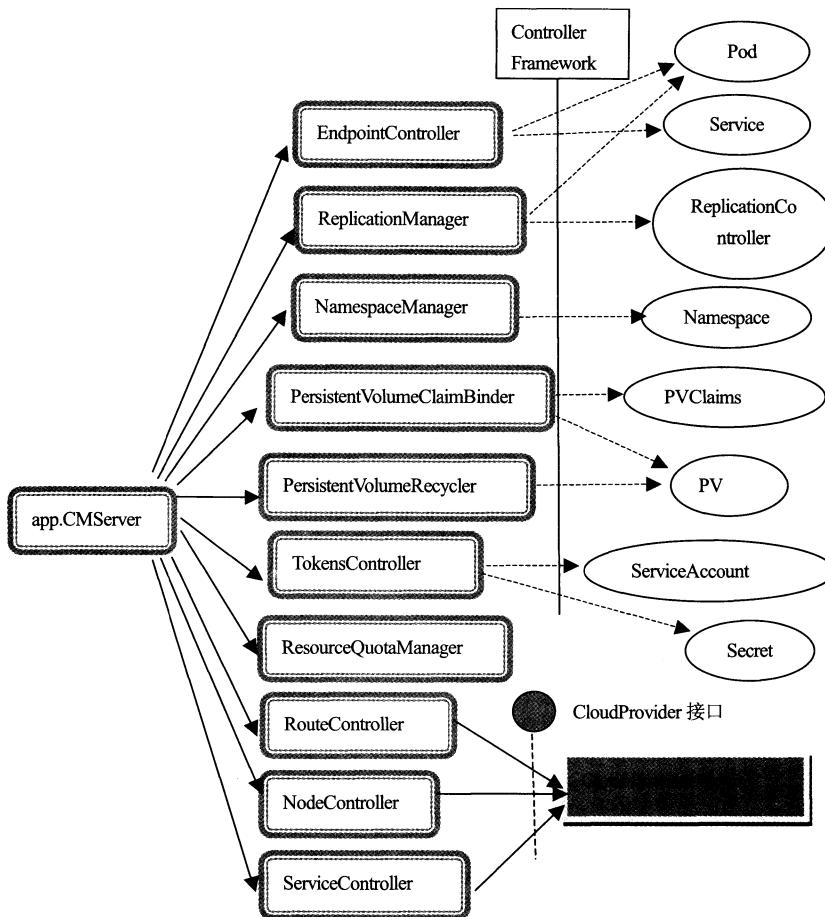


图 6.5 Kubernetes Controller Server 整体设计示意图

6.4 kube-scheduler 进程源码分析

Kubernetes Scheduler Server 是由 kube-scheduler 进程实现的，它运行在 Kubernetes 的管理节点——Master 上并主要负责完成从 Pod 到 Node 的调度过程。Kubernetes Scheduler Server 跟踪 Kubernetes 集群中所有 Node 的资源利用情况，并采取合适的调度策略，确保调度的均衡性，避免集群中的某些节点“过载”。从某种意义上来说，Kubernetes Scheduler Server 也是 Kubernetes 集群的“大脑”。

谷歌作为公有云的重要供应商，积累了很多经验并且了解客户的需求。在谷歌看来，客户

并不真正关心他们的服务究竟运行在哪台机器上，他们最关心服务的可靠性，希望发生故障后能自动恢复。遵循这一指导思想，Kubernetes Scheduler Server 实现了“完全市场经济”的调度原则并彻底抛弃了传统意义上的“计划经济”。

下面我们分别对其启动过程、关键代码分析及设计总结等方面进行深入分析和讲解。

6.4.1 进程启动过程

kube-scheduler 进程的入口类源码位置如下：

[github.com/GoogleCloudPlatform/kubernetes/plugin/cmd/kube-scheduler/scheduler.go](https://github.com/GoogleCloudPlatform/kubernetes/blob/master/plugin/cmd/kube-scheduler/scheduler.go)。

入口 main() 函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewSchedulerServer()
    s.AddFlags(pflag.CommandLine)
    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()
    verflag.PrintAndExitIfRequested()
    s.Run(pflag.CommandLine.Args())
}
```

对上述代码的风格和逻辑我们再熟悉不过了：创建一个 SchedulerServer 对象，将命令行参数传入，并且进入 SchedulerServer 的 Run 方法，无限循环下去。

按照惯例，我们首先看看 SchedulerServer 的数据结构 ([app/server.go](#))，下面是其定义：

```
type SchedulerServer struct {
    Port          int
    Address       util.IP
    AlgorithmProvider string
    PolicyConfigFile string
    EnableProfiling bool
    Master         string
    Kubeconfig     string
}
```

这里的关键属性有以下两个。

- ◎ **AlgorithmProvider**: 对应参数 `algorithm-provider`，是 `AlgorithmProviderConfig` 的名称。
- ◎ **PolicyConfigFile**: 用来加载调度策略文件。

从代码上来看这两个参数的作用其实是一样的，都是加载一组调度规则，这组调度规则要

么在程序里定义为一个 AlgorithmProviderConfig，要么保存到文件中。下面的源码清楚地解释了这个过程：

```
func (s *SchedulerServer) createConfig(configFactory *factory.ConfigFactory)
(*scheduler.Config, error) {
    var policy schedulerapi.Policy
    var configData []byte

    if _, err := os.Stat(s.PolicyConfigFile); err == nil {
        configData, err = ioutil.ReadFile(s.PolicyConfigFile)
        if err != nil {
            return nil, fmt.Errorf("Unable to read policy config: %v", err)
        }
        err = latestschedulerapi.Codec.DecodeInto(configData, &policy)
        if err != nil {
            return nil, fmt.Errorf("Invalid configuration: %v", err)
        }
    }

    return configFactory.CreateFromConfig(policy)
}

// if the config file isn't provided, use the specified (or default) provider
// check of algorithm provider is registered and fail fast
_, err := factory.GetAlgorithmProvider(s.AlgorithmProvider)
if err != nil {
    return nil, err
}

return configFactory.CreateFromProvider(s.AlgorithmProvider)
}
```

创建了 SchedulerServer 结构体实例后，调用此实例的方法 func (s *APIServer) Run(_[]string)，进入关键流程。首先，创建一个 Rest Client 对象用于访问 Kubernetes API Server 提供的 API 服务：

```
kubeClient, err := client.New(kubeconfig)
if err != nil {
    glog.Fatalf("Invalid API configuration: %v", err)
}
```

随后，创建一个 HTTP Server 以提供必要的性能分析（Performance Profile）和性能指标度量（Metrics）的 Rest 服务：

```
go func() {
    mux := http.NewServeMux()
    healthz.InstallHandler(mux)
    if s.EnableProfiling {
        mux.HandleFunc("/debug/pprof/", pprof.Index)
        mux.HandleFunc("/debug/pprof/profile", pprof.Profile)
    }
}
```

```

        mux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
    }
    mux.Handle("/metrics", prometheus.Handler())

    server := &http.Server{
        Addr:   net.JoinHostPort(s.Address.String(), strconv.Itoa(s.Port)),
        Handler: mux,
    }
    glog.Fatal(server.ListenAndServe())
}()

```

接下来，启动程序构造了 ConfigFactory，这个结构体包括了创建一个 Scheduler 所需的必要属性。

- ◎ PodQueue：需要调度的 Pod 队列。
- ◎ BindPodsRateLimiter：调度过程中限制 Pod 绑定速度的限速器。
- ◎ modeler：这是用于优化 Pod 调度过程而设计的一个特殊对象，用于“预测未来”。一个 Pod 被计划调度到机器 A 的事实被称为 assumed 调度，即假定调度，这些调度安排被保存到特定队列里，此时调度过程是能看到这个预安排的，因而会影响到其他 Pod 的调度。
- ◎ PodLister：负责拉取已经调度过的，以及被假定调度过的 Pod 列表。
- ◎ NodeLister：负责拉取 Node 节点（Minion）列表。
- ◎ ServiceLister：负责拉取 Kubernetes 服务列表。
- ◎ ScheduledPodLister、scheduledPodPopulator：Controller 框架创建过程中返回的 Store 对象与 controller 对象，负责定期从 Kubernetes API Server 上拉取已经调度好的 Pod 列表，并将这些 Pod 从 modeler 的假定调度过的队列中删除。

在构造 ConfigFactory 的方法 factory.NewConfigFactory(kubeClient) 中，我们看到下面这段代码：

```

c.ScheduledPodLister.Store, c.scheduledPodPopulator = framework.NewInformer(
    c.createAssignedPodLW(),
    &api.Pod{},
    0,
    framework.ResourceEventHandlerFuncs{
        AddFunc: func(obj interface{}) {
            if pod, ok := obj.(*api.Pod); ok {
                c.modeler.LockedAction(func() {
                    c.modeler.ForgetPod(pod)
                })
            }
        },
    },
)

```

```
        DeleteFunc: func(obj interface{}) {
            c.modeler.LockedAction(func() {
                switch t := obj.(type) {
                    case *api.Pod:
                        c.modeler.ForgetPod(t)
                    case cache.DeletedFinalStateUnknown:
                        c.modeler.ForgetPodByKey(t.Key)
                }
            })
        },
    },
}
```

这里沿用了之前看到的 controller framework 的身影，上述 Controller 实例所做的事情是获取并监听已经调度的 Pod 列表，并将这些 Pod 列表从 modeler 中的“assumed”队列中删除。

接下来，启动进程用上述创建好的 ConfigFactory 对象作为参数来调用 SchdulerServer 的 createConfig 方法，创建一个 Scheduler.Config 对象，而此段代码的关键逻辑则集中在 ConfigFactory 的 CreateFromKeys 这个函数里，其主要步骤如下。

(1) 创建一个与 Pod 相关的 Reflector 对象并定期执行, 该 Reflector 负责查询并监测等待调度的 Pod 列表, 即还没有分配主机的 Pod (Unsigned Pod), 然后把它们放入 ConfigFactory 的 PodQueue 中等待调度。相关代码为: `cache.NewReflector(f.createUnassignedPodLW(), &api.Pod{}, f.PodQueue, 0).RunUntil(f.StopEverything)`。

(2) 启动 ConfigFactory 的 scheduledPodPopulator Controller 对象, 负责定期从 Kubernetes API Server 上拉取已经调度好的 Pod 列表, 并将这些 Pod 从 modeler 中的假定 (assumed) 调度过过的队列中删除。相关代码为: go f.scheduledPodPopulator.Run(f.StopEverything)。

(3) 创建一个 Node 相关的 Reflector 对象并定期执行, 该 Reflector 负责查询并监测可用的 Node 列表(可用意味着 Node 的 spec.unschedulable 属性为 false), 这些 Node 被放入 ConfigFactory 的 NodeLister.Store 里。相关代码为: `cache.NewReflector(f.createMinionLW(), &api.Node{}, f.NodeLister.Store, 0).RunUntil(f.StopEverything)`。

(4) 创建一个 Service 相关的 Reflector 对象并定期执行，该 Reflector 负责查询并监测已定义的 Service 列表，并放入 ConfigFactory 的 ServiceLister.Store 里。这个过程的目的是 Scheduler 需要知道一个 Service 当前所创建的所有 Pod，以便能正确地进行调度。相关代码为：cache.NewReflector(f.createServiceLW(), &api.Service{}, f.ServiceLister.Store, 0).RunUntil(f.StopEverything)。

(5) 创建一个实现了 `algorithm.ScheduleAlgorithm` 接口的对象 `genericScheduler`, 它负责完成从 Pod 到 Node 的具体调度工作, 调度完成的 Pod 放入 `ConfigFactory` 的 `PodLister` 里。相关代码

码为 algo := scheduler.NewGenericScheduler(predicateFuncs, priorityConfigs, f.PodLister, r)。

(6) 最后一步，使用之前的这些信息创建 Scheduler.Config 对象并返回。

从上面的分析我们看出，其实在创建 Scheduler.Config 的过程中已经完成了 Kubernetes Scheduler Server 进程中的很多启动工作，于是整个进程的启动过程的最后一步简单明了：使用刚刚创建好的 Config 对象来构造一个 Scheduler 对象并启动运行。即下面的两行代码：

```
sched := scheduler.New(config)
sched.Run()
```

而 Scheduler 的 Run 方法就是不停地执行 scheduleOne 方法：

```
go util.Until(s.scheduleOne, 0, s.config.StopEverything)
```

scheduleOne 方法的逻辑也比较清晰，即获取下一个待调度的 Pod，然后交给 genericScheduler 进行调度（完成 Pod 到某个 Node 的绑定过程），调度成功以后通知 Modeler。这个过程同时增加了限流和性能指标的逻辑。

6.4.2 关键代码分析

在 6.4.1 节对 kube-scheduler 进程的启动过程进行详细分析后，我们大致明白了 Kubernetes Scheduler Server 的工作流程，但由于代码中涉及多个 Pod 队列和 Pod 状态切换逻辑，因此这里有必要对这个问题进行详细分析，以弄清在整个调度过程中 Pod 的“来龙去脉”。首先，我们知道 ConfigFactory 里的 PodQueue 是“待调度的 Pod 队列”，这个过程是通过无限循环执行一个 Reflector 来从 Kubernetes API Server 上获取待调度的 Pod 列表并填充到队列中实现的，因为 Reflector 框架已经实现了通用的代码，所以到了 Kubernetes Scheduler Server 这里，通过一行代码就能完成这个复杂的过程：

```
cache.NewReflector(f.createUnassignedPodLW(), &api.Pod{}, f.PodQueue, 0).
RunUntil(f.StopEverything)
```

上述代码中的 createUnassignedPodLW 是查询和监测 spec.nodeName 为空的 Pod 列表，此外，我们注意到 scheduler.Config 里提供了 NextPod 这个函数指针来从上述队列中消费一个元素，下面是相关代码片段(来自 ConfigFactory 的 CreateFromKeys 方法中创建 scheduler.Config 的代码)：

```
NextPod: func() *api.Pod {
    pod := f.PodQueue.Pop().(*api.Pod)
    glog.V(2).Infof("About to try and schedule pod %v", pod.Name)
    return pod
},
```

然后，这个 PodQueue 是怎样被消费的呢？就在之前提到的 Scheduler.scheduleOne 的方法里，每次调用 NextPod 方法会获取一个可用的 Pod，然后交给 genericScheduler 进行调度，下面是相

关代码片段（省略了其他代码）：

```
pod := s.config.NextPod()
if s.config.BindPodsRateLimiter != nil {
    s.config.BindPodsRateLimiter.Accept()
}
dest, err := s.config.Algorithm.Schedule(pod, s.config.MinionLister)
```

genericScheduler.Schedule 方法只是给出该 Pod 调度到的目标 Node，如果调度成功，则设置该 Pod 的 spec.nodeName 为目标 Node，然后通过 HTTP Rest 调用写入 Kubernetes API Server 里完成 Pod 的 Binding 操作，最后通知 ConfigFactory 的 modeler（具体实例对应 scheduler.SimpleModeler），将此 Pod 放入 Assumed Pod 队列，下面是相关代码片段：

```
s.config.Modeler.LockedAction(func() {
    bindingStart := time.Now()
    err := s.config.Binder.Bind(b)

    metrics.BindingLatency.Observe(metrics.SinceInMicroseconds(bindingStart))
    s.config.Recorder.Eventf(pod, "scheduled", "Successfully assigned %v to
%v", pod.Name, dest)
    // tell the model to assume that this binding took effect.
    assumed := *pod
    assumed.Spec.NodeName = dest
    s.config.Modeler.AssumePod(&assumed)
})
```

当 Pod 执行 Bind 操作成功以后，Kubernetes API Server 上 Pod 已经满足“已调度”的条件，因为 spec.nodeName 已经被设置为目标 Node 地址，此时 ConfigFactory 的 scheduledPodPopulator 这个 Controller 就会监听到此变化，将此 Pod 从 modeler 中的 Assumed 队列中删除，下面是相关代码片段：

```
framework.ResourceEventHandlerFuncs{
    AddFunc: func(obj interface{}) {
        if pod, ok := obj.(*api.Pod); ok {
            c.modeler.LockedAction(func() {
                c.modeler.ForgetPod(pod)
            })
        }
    },
    .....
},
```

谷歌的大神在源码中说明 Modeler 的存在是为了调度的优化，那么这个优化具体体现在哪里呢？由于 Rest Watch API 存在延时，当前已经调度好的 Pod 很可能还未被通知给 Scheduler，于是大神灵光一闪：为每个刚刚调度完成的 Pod 发放一个“暂住证”，安排“暂住”到“Assumed”队列里，然后设计一个获取当前“已调度”的 Pod 队列的新方法，该方法合并 Assumed 队列与

Watch 缓存队列，这样一来，就得到了最佳答案。如果你打算看看这段代码，那么它就在 SimpleModeler 的 listPods 方法里，至此，你若也完全明白了 c.PodLister = modeler.PodLister() 这句简单却又深奥的代码，那么恭喜你，你离大神的距离又缩短了一个厘米。

接下来，我们深入分析 Pod 调度中所用到的流控技术，缘起于下面这段代码：

```
if s.config.BindPodsRateLimiter != nil {
    s.config.BindPodsRateLimiter.Accept()
}
```

上述代码中的 BindPodsRateLimiter 采用了开源项目 juju 的一个子项目 ratelimit，项目地址为 <https://github.com/juju/ratelimit>，它实现了一个高效的基于经典令牌桶（Token Bucket）的流控算法。如图 6.6 所示是经典令牌桶流控算法示意图。

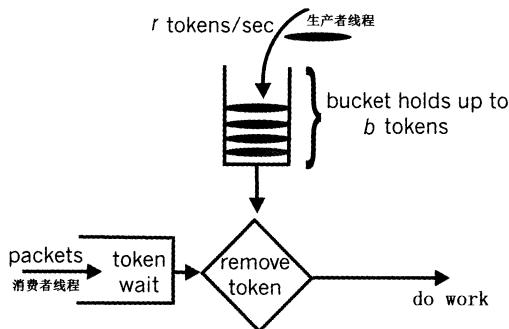


图 6.6 令牌桶流控算法示意图

简单地说，控制线程以固定速率向一个固定容量的桶（Bucket）中投放令牌（Token），消费者线程则等待并获取到一个令牌后才能继续接下来的任务，否则需要等待可用令牌的到来。具体说来，假如用户配置的平均限流速率为 r ，则每隔 $1/r$ 秒就会有一个令牌被加入桶中，而令牌桶最多可以存储 b 个令牌，如果令牌到达时令牌桶已经满了，那么这个令牌会被丢弃。从长期运行结果来看，消费者的处理速率被限制成常量 r 。令牌桶流控算法除了能够限制平均处理速度，还允许某种程度的突发速率。

juju 的 ratelimit 模块通过下面的 API 提供了构造一个令牌桶的简单做法，其中，rate 参数表示每秒填充到桶里的令牌数量，capacity 则是桶的容量：

```
func NewBucketWithRate(rate float64, capacity int64) *Bucket
```

我们回头再看看 Kubernetes Scheduler Server 中 BindPodsRateLimiter 的赋值代码：c.BindPodsRateLimiter = util.NewTokenBucketRateLimiter(bindPodsQps, bindPodsBurst)，跟踪进去，发现它就是调用了刚才所提到的 juju 函数 limiter := ratelimit.NewBucketWithRate(float64(qps), int64(burst))，其中 qps 目前为常量 15，而 burst 为 20，目前在 Kubernetes 1.0 版本中还没有提供命令行参数来配置此变量，会在未来的版本中实现。

最后，我们一起深入分析 Kubernetes Scheduler Server 中关于 Pod 调度的细节。首先，我们需要理解启动过程中 SchedulerServer 加载调度策略相关配置的这段代码：

```
predicateFuncs, err := getFitPredicateFunctions(predicateKeys, pluginArgs)
priorityConfigs, err := getPriorityFunctionConfigs(priorityKeys, pluginArgs)
algo := scheduler.NewGenericScheduler(predicateFuncs, priorityConfigs, f.
PodLister, r)
```

这里加载了两组策略，其中 predicateFuncs 是一个 Map，key 为 FitPredicate 的名称，value 为对应的 algorithm.FitPredicate 函数，它表明一个候选的 Node 是否满足当前 Pod 的调度要求，FitPredicate 函数的具体定义如下：

```
type FitPredicate func(pod *api.Pod, existingPods []*api.Pod, node string) (bool,
error)
```

FitPredicate 是 Pod 调度过程中必须满足的规则，只有顺利通过由所有 FitPredicate 组成的这道封锁线，一个 Node 才能拿到主会场的“入场券”，成为一个合格的“候选人”，等待下一步“评审”。目前系统提供的具体的 FitPredicate 实现都在 predicates.go 里，系统默认加载注册 FitPredicate 的地方在 defaultPredicates 方法里。

当有一组 Node 通过筛查成为“候选人”之后，需要有一种办法来选择“最优”的 Node，这就是接下来我们要介绍的 priorityConfigs 所要做的事情了。priorityConfigs 是一个数组，类型为 algorithm.PriorityConfig，PriorityConfig 包括一个 PriorityFunction 函数，用来计算并给出一组 Node 的优先级，下面是相关代码：

```
type PriorityConfig struct {
    Function PriorityFunction
    Weight   int
}

type PriorityFunction func(pod *api.Pod, podLister PodLister, minionLister
MinionLister) (HostPriorityList, error)

type HostPriorityList []HostPriority
func (h HostPriorityList) Len() int {
    return len(h)
}
func (h HostPriorityList) Less(i, j int) bool {
    if h[i].Score == h[j].Score {
        return h[i].Host < h[j].Host
    }
    return h[i].Score < h[j].Score
}
```

如果看到这里还是不太明白它的用途，那么认真读一读下面这段来自 genericScheduler 的计算候选节点优先级的 PrioritizeNodes 方法，你就能顿悟了：一个候选节点的优先级总分是所有评委老师(PriorityConfig)一起给出的“加权总分”，评委老师越是德高望重(PriorityConfig.Weight

越大），他的评分影响力就越大：

```
combinedScores := map[string]int{}
for _, priorityConfig := range priorityConfigs {
    weight := priorityConfig.Weight
    // skip the priority function if the weight is specified as 0
    if weight == 0 {
        continue
    }
    priorityFunc := priorityConfig.Function
    prioritizedList, err := priorityFunc(pod, podLister, minionLister)
    if err != nil {
        return algorithm.HostPriorityList{}, err
    }
    for _, hostEntry := range prioritizedList {
        combinedScores[hostEntry.Host] += hostEntry.Score * weight
    }
}
for host, score := range combinedScores {
    glog.V(10).Infof("Host %s Score %d", host, score)
    result = append(result, algorithm.HostPriority{Host: host, Score: score})
}
return result, nil
```

接下来，我们看看系统初始化加载的默认的 Predicate 与 Priorities 有哪些，通过追踪代码，我们发现默认加载的代码位于 plugin/pkg/scheduler/algorithmprovider/default/default.go 的 init 函数里：

```
func init() {
    factory.RegisterAlgorithmProvider(factory.DefaultProvider, defaultPredicates(),
    defaultPriorities())
    // EqualPriority is a prioritizer function that gives an equal weight of one
    to all minions
    // Register the priority function so that its available
    // but do not include it as part of the default priorities
    factory.RegisterPriorityFunction("EqualPriority", scheduler.EqualPriority, 1)
}
```

跟踪进去后，我们看到系统默认加载的 predicates 有如下几种：

- ◎ PodFitsResources;
- ◎ MatchNodeSelector;
- ◎ HostName。

而默认加载的 priorities 则有如下几种：

- ◎ LeastRequestedPriority;
- ◎ BalancedResourceAllocation;

◎ ServiceSpreadingPriority。

从上述这些信息来看,Kubernetes 默认的调度指导原则是尽量均匀分布 Node 到不同的 Node 上,并且确保各个 Node 上的资源利用率基本保持一致,也就是说如果你有 100 台机器,则可能每个机器都被调度到,而不是只有其中的 20% 被调度到,哪怕每台机器都只利用了不到 10% 的资源,这不正是所谓的“韩信点兵,多多益善”么?

接下来我们以服务亲和性这个默认没有加载的 Predicate 为例,看看 Kubernetes 是如何通过 Policy 文件注册加载它的。下面是我们定义的一个 Policy 文件:

```
{
  "kind" : "Policy",
  "version" : "v1",
  "predicates" : [
    .....
    {"name" : "RegionZoneAffinity", "argument" : {"serviceAffinity" :
  "labels" : ["region", "zone"]}}}
  ],
  "priorities" : [
    .....
    {"name" : "RackSpread", "weight" : 1, "argument" : {"serviceAnti
  Affinity" : {"label" : "rack"}}}
  ]
}
```

首先,这个文件被映射成 api.Policy 对象 (plugin/pkg/scheduler/api/types.go)。下面是其结构体定义:

```
type Policy struct {
  api.TypeMeta `json: ",inline"`
  // Holds the information to configure the fit predicate functions
  Predicates []PredicatePolicy `json: "predicates"`
  // Holds the information to configure the priority functions
  Priorities []PriorityPolicy `json: "priorities"`
}
```

我们看到 policy 文件中的 predicates 部分被映射为 PredicatePolicy 数组:

```
type PredicatePolicy struct {
  Name string `json: "name"`
  Argument *PredicateArgument `json: "argument"`
}
```

而 PredicateArgument 的定义如下,包括服务亲和性的相关属性 ServiceAffinity:

```
type PredicateArgument struct {
  ServiceAffinity *ServiceAffinity `json: "serviceAffinity"`
  LabelsPresence *LabelsPresence `json: "labelsPresence"`
}
```

策略文件被映射为 `api.Policy` 对象后，`PredicatePolicy` 部分的处理逻辑则交给下面的函数进行处理 (`plugin/pkg/scheduler/factory/plugin.go`)：

```
func RegisterCustomFitPredicate(policy schedulerapi.PredicatePolicy) string {
    var predicateFactory FitPredicateFactory
    var ok bool
    validatePredicateOrDie(policy)
    // generate the predicate function, if a custom type is requested
    if policy.Argument != nil {
        if policy.Argument.ServiceAffinity != nil {
            predicateFactory = func(args PluginFactoryArgs) algorithm.FitPredicate {
                return predicates.NewServiceAffinityPredicate(
                    args.PodLister,
                    args.ServiceLister,
                    args.NodeInfo,
                    policy.Argument.ServiceAffinity.Labels,
                )
            }
        } else if policy.Argument.LabelsPresence != nil {
            predicateFactory = func(args PluginFactoryArgs) algorithm.FitPredicate {
                return predicates.NewNodeLabelPredicate(
                    args.NodeInfo,
                    policy.Argument.LabelsPresence.Labels,
                    policy.Argument.LabelsPresence.Presence,
                )
            }
        }
    }
}
```

在上面的代码中，当 `ServiceAffinity` 属性不空时，就会调用 `predicates.NewServiceAffinityPredicate` 方法来创建一个处理服务亲和性的 `FitPredicate`，随后被加载到全局的 `predicateFactory` 中生效。

最后，`genericScheduler.Schedule` 方法才是真正实现 Pod 调度的方法，我们看看这段完整代码：

```
func (g *genericScheduler) Schedule(pod *api.Pod, minionLister algorithm.MinionLister) (string, error) {
    minions, err := minionLister.List()
    if err != nil {
        return "", err
    }
    if len(minions.Items) == 0 {
        return "", ErrNoNodesAvailable
    }

    filteredNodes, failedPredicateMap, err := findNodesThatFit(pod, g.pods,
        g.predicates, minions)
    if err != nil {
```

```

        return "", err
    }

    priorityList, err := PrioritizeNodes(pod, g.pods, g.prioritizers, algorithm.
FakeMinionLister(filteredNodes))
    if err != nil {
        return "", err
    }
    if len(priorityList) == 0 {
        return "", &FitError{
            Pod:           pod,
            FailedPredicates: failedPredicateMap,
        }
    }

    return g.selectHost(priorityList)
}

```

这段代码已经简单得不能再简单了，因为该干的活都已经被 predicates 与 priorities 干完了！架构之美，就在于程序逻辑分解得恰到好处，每个组件各司其职，从而化繁为简，使得主体流程清晰直观，犹如行云流水，一气呵成。

向谷歌大神们致敬！

6.4.3 设计总结

与之前的 Kubernetes API Server 和 Kubernetes Controller Manager 对比，Kubernetes Scheduler Server 的设计和代码显得更为“精妙”。项目中引入 ratelimit 组件来解决 Pod 调度的流控问题的做法，既大大简化了代码量，又体现了大神们的气度。

Kubernetes Scheduler Server 的一个关键设计目标是“插件化”，以方便 Cloud Provider 或者个人用户根据自己的需求进行定制，本节我们围绕其中最为关键的“FitPredicate 与 PriorityFunction”对其设计做一个总结。如图 6.7 所示，在 plugin.go 中采用了全局变量的 Map 变量记录了系统当前注册的 FitPredicate 与 PriorityFunction，其中 fitPredicateMap 和 priorityFunctionMap 分别存放 FitPredicateFactory 与 PriorityConfigFactory（包含了 PriorityFunctionFactory 的一个引用）中。可以看出，这里的设计采用了标准的工厂模式，factory.PluginFactoryArgs 这个数据结构可以认为是一个上下文环境变量，它提供给 PluginFactory 必要的数据访问接口，比如获取一个 Node 的详细信息并获取一个 Pod 上的所有 Service 信息等，这些接口可以被某些具体的 FitPredicate 或 PriorityFunction 使用，以实现特定的功能，如图 6.7 所示的 predicates.PodFitsPods 和 priorities.LeastRequestedPriority 就分别使用了上述接口。

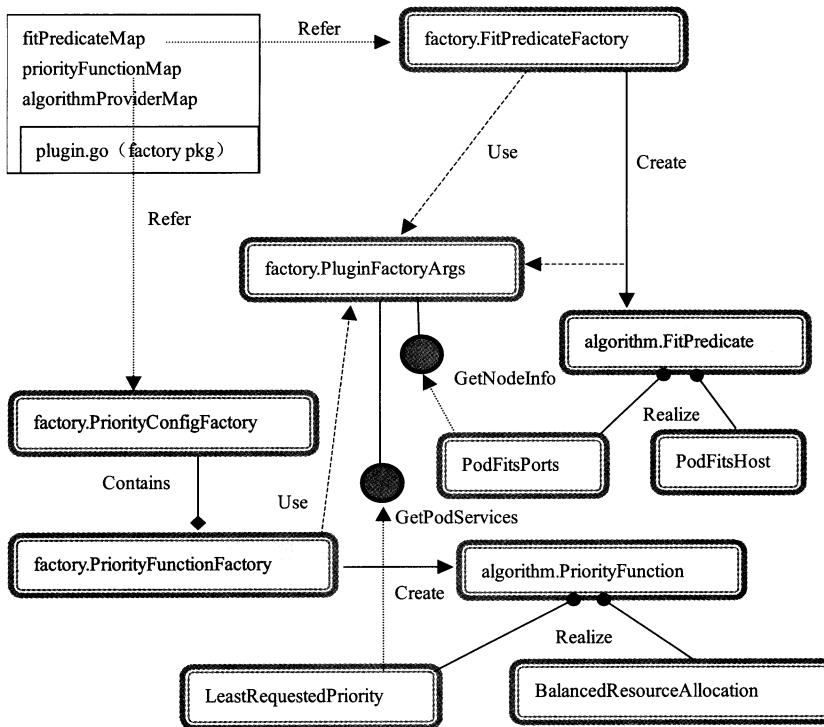


图 6.7 Kubernetes Scheduler Server 调度策略相关设计示意图

我们注意到 PluginFactoryArgs 的接口都是 Kubernetes 的资源访问接口，那么问题就来了，为何不直接用 Kubernetes RestClient API 访问呢？一个主要的原因是如果这样做，则增加了插件开发者开发和调测的难度，因为开发者需要再去学习和掌握 RestClient；另外一个原因是效率的问题，如果大家都采用框架提供的“标准方法”查询资源，那么框架可以实现很多优化，比较容易缓存；最后一个原因则与之前我们分析的“Assumed Pod”有关，即查询当前已经调度过的 Pod 列表是有其特殊性的，PluginFactoryArgs 中的 PodLister 方法就是引用了 ConfigFactory 的 PodLister。

algorithmProviderMap 这个全局变量则保存了一组命名的调度策略配置文件（AlgorithmProviderConfig），其实就是一组 FitPredicate 与 PriorityFunction 的集合，其定义如下：

```

type AlgorithmProviderConfig struct {
    FitPredicateKeys    util.StringSet
    PriorityFunctionKeys util.StringSet
}

```

它的作用是预配置和自定义调度规则，Kubernetes Scheduler Server 默认加载了一个名为“DefaultProvider”的调度策略配置，通过定义和加载不同的调度规则配置文件，我们可以改变默认的调度策略，比如我们可以定义两组规则文件：其中一个命名为“function_test_cfg”，面向

功能测试，调度原则是尽量在最少的机器上调度 Pod 以节省资源；另外一个则命名为 `performance_test_cfg`，面向性能测试，调度原则是尽可能使用更多的机器，以测试系统性能。

顺便提一下，笔者认为在 `Kubernetes Scheduler Server` 中关于 `PredicateArgument/Priority Argument` 的设计并不好，这里没有将 `Predicate` 的属性通用化，比如采用 `key-value` 这种模式，因此导致 `Policy` 文件格式与 `Predicate/Priority` 关联之间的强耦合性，增加了代码理解的困难性，之前分析的 `Policy` 文件中服务亲和性的 `Predicate` 的加载逻辑即反映了这个问题，笔者深信，未来版本中大神们会认真考虑重构问题。

至此，Master 节点上的进程的源码都已经分析完毕，我们发现这些进程所做的事情，归根到底就是两件事：Pod 调度+智能纠错，这也是为什么这些进程所在的节点被称为“Master”，因为它们高高在上，运筹帷幄。虽然“Master”从不深入底层微服私访，但也的确鞠躬尽瘁、日理万机，计算机的世界果然比我们人类的世界要单纯、高效很多，真心希望人工智能的发展不会让它们的世界也变得扑朔迷离。

6.5 kubelet 进程源码分析

`kubelet` 是运行在 Minion 节点上的重要守护进程，是工作在一线的重要“工人”，它才是负责“实例化”和“启动”一个具体的 Pod 的幕后主导，并且掌管着本节点上的 Pod 和容器的全生命周期过程，定时向 Master 汇报工作情况。此外，`kubelet` 进程也是一个“Server”进程，它默认监听 10250 端口，接收并执行远程（Master）发来的指令。

下面我们分别对其启动过程、关键代码分析及设计总结等方面进行深入分析和讲解。

6.5.1 进程启动过程

`kubelet` 进程的入口类源码位置如下：

[github.com/GoogleCloudPlatform/kubernetes/cmd/kubelet/kubelet.go](https://github.com/GoogleCloudPlatform/kubernetes/blob/master/cmd/kubelet/kubelet.go)

入口 `main()` 函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewKubeletServer()
    s.AddFlags(pflag.CommandLine)
    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()
```

```
verflag.PrintAndExitIfRequested()
if err := s.Run(pflag.CommandLine.Args()); err != nil {
    fmt.Fprintf(os.Stderr, "%v\n", err)
    os.Exit(1)
}
}
```

我们已经是第 4 次“遇见”这样的代码风格了，代码的颜值匹配度高达 99%，这至少说明一点：谷歌在源码一致性方面做得很好， N 多人写的代码看起来就好像出自一个人之手。我们先来看看 KubeletServer 这个结构体所包括的属性吧，这些属性可以分为以下几组。

1) 基本配置

- ◎ KubeConfig: kubelet 默认配置文件路径。
- ◎ Address、Port、ReadOnlyPort、CadvisorPort、HealthzPort、HealthzBindAddress: 为 kubelet 绑定监听的地址，包括自身 Server 的地址，Cadvisor 绑定的地址，以及自身健康检查服务的绑定地址等。
- ◎ RootDirectory、CertDirectory: kubelet 默认的工作目录 (/var/lib/kubelet)，用于存放配置及 VM 卷等数据，CertDirectory 用于存放证书目录。

2) 管理 Pod 和容器相关的参数

- ◎ PodInfraContainerImage: Pod 的 infra 容器的镜像名称，谷歌被屏蔽的时候可以换成自己的私有仓库的镜像名。
- ◎ CgroupRoot: 可选项，创建 Pod 的时候所使用的顶层的 cgroup 名字 (Root Cgroup)。
- ◎ ContainerRuntime、DockerDaemonContainer、SystemContainer: 这三个参数分别表示选择什么容器技术 (Docker 或者 RKT)、Docker Daemon 容器的名字及可选的系统资源容器名称，用来将所有非 kernel 的、不在容器中的进程放入此容器中。

3) 同步和自动运维相关的参数

- ◎ SyncFrequency、FileCheckFrequency、HTTPCheckFrequency: Pod 容器同步周期、当前运行的容器实例分别与 Kubernetes 注册表中的信息、本地的 Pod 定义文件及以 HTTP 方式提供信息的数据源进行对比同步。
- ◎ RegistryPullQPS、RegistryBurst: 从注册表拉取待创建的 Pod 列表时的流控参数。
- ◎ NodeStatusUpdateFrequency: kubelet 多久汇报一次当前 Node 的状态。
- ◎ ImageGCHighThresholdPercent、ImageGCLowThresholdPercent、LowDiskSpaceThresholdMB: 分别是 Image 镜像占用磁盘空间的高低水位阈值及本机磁盘最小空闲容量，当可用容量低于这个容量时，所有新 Pod 的创建请求会被拒绝。

- ◎ MaxContainerCount、MaxPerPodContainerCount：分别是 maximum-dead-containers 与 maximum-dead-containers-per-container，表示保留多少个死亡容器的实例在磁盘上，因为每个实例都会占用一定的磁盘，所以需要控制，默认是 MaxContainerCount 为 100，MaxPerPodContainerCount 为 2，即每个容器保留最多两个死亡实例，每个 Node 保留最多 100 个死亡实例。

只要分析一下上述 KubeletServer 结构体的关键属性，我们就可以得到这样一个推论：kubelet 进程的“工作量”还是很饱满的，一点都不比 Master 上的 API Server、Controll Manager、Scheduer 做得少。

在继续下面的代码分析之前，我们先要理解这里的一个重要概念“Pod Source”，它是 kubelet 用于获取 Pod 定义和描述信息的一个“数据源”，kubelet 进程查询并监听 Pod Source 来获取属于自己所在节点的 Pod 列表，当前支持三种 Pod Source 类型。

- ◎ Config File：本地配置文件作为 Pod 数据源。
- ◎ Http URL：Pod 数据源的内容通过一个 HTTP URL 方式获取。
- ◎ Kubernetes API Server：默认方式，从 API Server 获取 Pod 数据源。

进程根据启动参数创建了 KubeletServer 以后，调用 KubeletServer 的 run 方法，进入启动流程，在流程的一开始首先设置了自身进程的 oom_adj 参数（默认为-900），这是利用了 Linux 的 OOM 机制，当系统发生 OOM 时，oom_adj 的值越小，越不容易被系统 Kill 掉。

```
if err := util.ApplyOomScoreAdj(0, s.OOMScoreAdj); err != nil {
    glog.Warning(err)
}
```

为什么在之前的 Master 节点进程上都没有见到这个调用，而在 kubelet 进程上却看到这段逻辑？答案很简单，因为 Master 节点不运行 Pod 和容器，主机资源通常是稳定和宽裕的，而 Minion 节点由于需要运行大量的 Pod 和容器，因此容易产生 OOM 问题，所以这里要确保“守护者”不会因此而被系统 Kill 掉。

由于 kubelet 会跟 API Server 打交道，所以接下来创建了一个 Rest Client 对象来访问 API Server。随后，启动进程构造了 cAdvisor 来监控本地的 Docker 容器，cAdvisor 具体的创建代码则位于 pkg/kubelet/cadvisor/cadvisor_linux.go 里，引用了 github.com/google/cadvisor 这个同样属于谷歌开源的项目。

接着，初始化 CloudProvider，这是因为如果 Kubernetes 运行在某个运营商的 Cloud 环境中，则很多环境和资源都需要从 CloudProvider 中获取，比如在创建 Pod 的过程中可能需要知道某个 Node 的真实主机名。

虽然容器可以绑定宿主机的网络空间，但若不当使用会导致系统安全漏洞，所以

KubeletServer 中的 HostNetworkSources 的属性用来控制哪些 Pod 允许绑定宿主机的网络空间，默认是都禁止绑定。举例说明，比如设置 HostNetworkSources=api,http，则表明当一个 Pod 的定义源来自 Kubernetes API Server 或者某个 HTTP URL 时，则允许此 Pod 绑定到宿主机的网络空间。下面这行代码即上述处理逻辑中的一小部分：

```
hostNetworkSources, err :=  
kubelet.GetValidatedSources(strings.Split(s.HostNetworkSources, ","))
```

接下来加载数字证书，如果没有提供证书和私钥，则默认创建一个自签名的 X509 证书并保存到本地。下一步，创建一个 Mounter 对象，用来实现容器的文件系统挂载功能。

接下来的这段代码根据指定了 DockerExecHandlerName 参数的值，确定 dockerExecHandler 是采用 Docker 的 exec 命令还是 nsenter 来实现，默认采用了 Docker 的 exec 这种本地方式，Docker 从 1.3 版本开始提供了 exec 指令，为进入容器内部提供了更好的手段。

```
var dockerExecHandler dockertools.ExecHandler  
switch s.DockerExecHandlerName {  
case "native":  
    dockerExecHandler = &dockertools.NativeExecHandler{}  
case "nsenter":  
    dockerExecHandler = &dockertools.NsenterExecHandler{}  
default:  
    log.Warningf("Unknown Docker exec handler %q; defaulting to native",  
s.DockerExecHandlerName)  
    dockerExecHandler = &dockertools.NativeExecHandler{}  
}
```

运行至此，程序构造了一个 KubeletConfig 结构体，90% 的变量与之前的 KubeletServer 一样，这让代码长度增加了 20 多行！定睛一看，源码上有 TODO 注释：“它应该可能被合并到 KubeletServer 里……”，目测注释是另外一个大神添加的，这让笔者陷入了深深的思考：难道谷歌的绩效考评系统中也有恶俗的代码行数考核指标？

KubeletConfig 创建好以后作为参数调用 RunKubelet (&kcfg, nil) 方法，程序运行到这里，才真正进入流程的核心步骤。下面这段代码表明 kubelet 会把自己的事件通知 API Server：

```
eventBroadcaster := record.NewBroadcaster()  
kcfg.Recorder = eventBroadcaster.NewRecorder(api.EventSource{Component:  
"kubelet", Host: kcfg.NodeName})  
eventBroadcaster.StartLogging(glog.V(3).Infof)  
if kcfg.KubeClient != nil {  
    glog.V(4).Infof("Sending events to api server. ")  
    eventBroadcaster.StartRecordingToSink(kcfg.KubeClient.Events(""))  
} else {  
    glog.Warning("No api server defined - no events will be sent to API server.  
")  
}
```

接下来，启动进程进入关键函数 `createAndInitKubelet` 中，这里首先创建一个 `PodConfig` 对象，并根据启动参数中 `Pod Source` 参数是否提供，来创建相应类型的 `Pod Source` 对象，这些 `PodSource` 在各种协程中运行，拉取 `Pod` 信息并汇总输出到同一个 `Pod Channel` 中等待 `kubelet` 处理。创建 `PodConfig` 的具体代码如下：

```
func makePodSourceConfig(kc *KubeletConfig) *config.PodConfig {
    // source of all configuration
    cfg := config.NewPodConfig(config.PodConfigNotificationSnapshotAndUpdates,
        kc.Recorder)

    // define file config source
    if kc.ConfigFile != "" {
        glog.Infof("Adding manifest file: %v", kc.ConfigFile)
        config.NewSourceFile(kc.ConfigFile, kc.NodeName, kc.FileCheckFrequency,
            cfg.Channel(kubelet.FileSource))
    }

    // define url config source
    if kc.ManifestURL != "" {
        glog.Infof("Adding manifest url: %v", kc.ManifestURL)
        config.NewSourceURL(kc.ManifestURL, kc.NodeName, kc.HTTPCheckFrequency,
            cfg.Channel(kubelet.HTTPSource))
    }

    if kc.KubeClient != nil {
        glog.Infof("Watching apiserver")
        config.NewSourceApiserver(kc.KubeClient, kc.NodeName, cfg.Channel
            (kubelet.ApiserverSource))
    }
    return cfg
}
```

然后，创建一个 `kubelet` 并宣告它的诞生：

```
k, err = kubelet.NewMainKubelet(....)
k.BirthCry()
```

接着，触发 `kubelet` 开启垃圾回收协程以清理无用的容器和镜像，释放磁盘空间，下面是其代码片段：

```
// Starts garbage collection threads.
func (kl *Kubelet) StartGarbageCollection() {
    go util.Forever(func() {
        if err := kl.containerGC.GarbageCollect(); err != nil {
            glog.Errorf("Container garbage collection failed: %v", err)
        }
    }, time.Minute)

    go util.Forever(func() {
```

```

        if err := kl.imageManager.GarbageCollect(); err != nil {
            glog.Errorf("Image garbage collection failed: %v", err)
        }
    }, 5*time.Minute)
}

```

`createAndInitKubelet` 方法创建 `kubelet` 实例以后，返回到 `RunKubelet` 方法里，接下来调用 `startKubelet` 方法，此方法首先启动一个协程，让 `kubelet` 处理来自 `PodSource` 的 `Pod Update` 消息，然后启动 `Kubelet Server`，下面是具体代码：

```

func startKubelet(k KubeletBootstrap, podCfg *config.PodConfig, kc *KubeletConfig) {
    // start the kubelet
    go util.Forever(func() { k.Run(podCfg.Updates()) }, 0)

    // start the kubelet server
    if kc.EnableServer {
        go util.Forever(func() {
            k.ListenAndServe(net.IP(kc.Address), kc.Port, kc.TLSOptions, kc.
EnableDebuggingHandlers)
        }, 0)
    }
    if kc.ReadOnlyPort > 0 {
        go util.Forever(func() {
            k.ListenAndServeReadonly(net.IP(kc.Address), kc.ReadOnlyPort)
        }, 0)
    }
}

```

至此，`kubelet` 进程启动完毕。

6.5.2 关键代码分析

6.5.1 节里，我们分析了 `kubelet` 进程的启动流程，大致明白了 `kubelet` 的核心工作流程就是不断从 `Pod Source` 中获取与本节点相关的 `Pod`，然后开始“加工处理”，所以，我们先来分析 `Pod Source` 部分的代码。前面我们提到，`kubelet` 可以同时支持三类 `Pod Source`，为了能够将不同的 `Pod Source` “汇聚”到一起统一处理，谷歌特地设计了 `PodConfig` 这个对象，其代码如下：

```

type PodConfig struct {
    pods *podStorage
    mux  *config.Mux

    // the channel of denormalized changes passed to listeners
    updates chan kubelet.PodUpdate

    // contains the list of all configured sources
}

```

```

sourcesLock sync.Mutex
sources     util.StringSet
}

```

其中，`sources` 属性包括了当前加载的所有 Pod Source 类型，`sourcesLock` 是 source 的排他锁，在新增 Pod Source 的方法里使用它来避免共享冲突。

当 Pod 发生变动时，例如 Pod 创建、删除或者更新，相关的 Pod Source 就会产生对应的 PodUpdate 事件并推送到 Channel 上。为了能够统一处理来自多个 Source 的 Channel，谷歌设计了 config.Mux 这个“聚合器”，它负责监听多路 Channel，当接收到 Channel 发来的事件以后，交给 Merger 对象进行统一处理，Merger 对象最终把多路 Channel 发来的事件合并写入 updates 这个汇聚 Channel 里等待处理。

下面是 config.Mux 的结构体定义，其属性 `sources` 为一个 Channel Map，key 是对应的 Pod Source 的类型：

```

type Mux struct {
    // Invoked when an update is sent to a source.
    merger Merger
    // Sources and their lock.
    sourceLock sync.RWMutex
    // Maps source names to channels
    sources map[string]chan interface{}
}

```

我们继续深入分析 config.Mux 的工作过程，前面提到，kubelet 在启动过程中在 makePodSourceConfig 方法里创建了一个 PodConfig 对象，并且根据启动参数来决定要加载哪些类型的 Pod Source，在这个过程中调用了下述方法来创建一个对应的 Channel：

```

func (c *PodConfig) Channel(source string) chan interface{} {
    c.sourcesLock.Lock()
    defer c.sourcesLock.Unlock()
    c.sources.Insert(source)
    return c.mux.Channel(source)
}

```

而 Channel 具体的创建过程则在 config.Mux 里，Channel 创建完成后被加入 config.Mux 的 `sources` 里并且启动一个协程开始监听消息，代码如下：

```

func (m *Mux) Channel(source string) chan interface{} {
    if len(source) == 0 {
        panic("Channel given an empty name")
    }
    m.sourceLock.Lock()
    defer m.sourceLock.Unlock()
    channel, exists := m.sources[source]
    if exists {

```

```

        return channel
    }
    newChannel := make(chan interface{})
    m.sources[source] = newChannel
    go util.Forever(func() { m.listen(source, newChannel) }, 0)
    return newChannel
}

```

`config.Mux` 的上述 `listen` 方法很简单，就是监听新创建的 `Channel`，一旦发现 `Channel` 上有数据就交给 `Merger` 进行处理：

```

func (m *Mux) listen(source string, listenChannel <-chan interface{}) {
    for update := range listenChannel {
        m.merger.Merge(source, update)
    }
}

```

我们先来看看 Pod Source 是如何发送 PodUpdate 事件到自己所在的 Channel 上的，在 6.5.1 节中我们所见到的下面这段代码创建了一个 Config File 类型的 Pod Source：

```

// define file config source
if kc.ConfigFile != "" {
    glog.Infof("Adding manifest file: %v", kc.ConfigFile)
    config.NewSourceFile(kc.ConfigFile, kc.NodeName, kc.FileCheckFrequency,
cfg.Channel(kubelet.FileSource))
}

```

在 `NewSourceFile` 方法里启动了一个协程，每隔指定的时间 (`kc.FileCheckFrequency`) 就执行一次 `SourceFile` 的 `run` 方法，在 `run` 方法里所调用的主体逻辑是下面的函数：

```

func (s *sourceFile) extractFromPath() error {
    path := s.path
    statInfo, err := os.Stat(path)
    if err != nil {
        if !os.IsNotExist(err) {
            return err
        }
        // Emit an update with an empty PodList to allow FileSource to be marked
        as seen
        s.updates <- kubelet.PodUpdate{[]*api.Pod{}, kubelet.SET, kubelet.
        FileSource}
        return fmt.Errorf("path does not exist, ignoring")
    }

    switch {
    case statInfo.Mode().IsDir():
        pods, err := s.extractFromDir(path)
        if err != nil {
            return err
    }
}

```

```

    }
    s.updates <- kubelet.PodUpdate{pods, kubelet.SET, kubelet.FileSource}

    case statInfo.Mode().IsRegular():
        pod, err := s.extractFromFile(path)
        if err != nil {
            return err
        }
        s.updates <- kubelet.PodUpdate{[]*api.Pod{pod}, kubelet.SET, kubelet.
FileSource}

    default:
        return fmt.Errorf("path is not a directory or file")
    }

    return nil
}

```

看一眼上面的代码，我们就大致明白了 Config File 类型的 Pod Source 是如何工作的：它从指定的目录中加载多个 Pod 定义文件并转换为 Pod 列表或者加载单个 Pod 定义文件并转换为单个 Pod，然后生成对应的全量类型的 PodUpdate 事件并写入 Channel 中去。这里笔者也发现了代码命名的一个疏漏之处，SourceFile 的 updates 属性其实应该被命名为 update。其他两种 Pod Source 类型的代码解析就不在这里提及了。

接下来我们分析 Merger 对象，PodConfig 里的 Merger 对象其实是一个 config.podStorage 实例，它同时是 PodConfig 的 pods 属性的一个引用。podStorage 的源码位于 pkg/kubelet/config/config.go 里，其定义如下：

```

type podStorage struct {
    podLock sync.RWMutex
    // map of source name to pod name to pod reference
    pods map[string]map[string]*api.Pod
    mode PodConfigNotificationMode
    // ensures that updates are delivered in strict order
    // on the updates channel
    updateLock sync.Mutex
    updates chan<- kubelet.PodUpdate
    // contains the set of all sources that have sent at least one SET
    sourcesSeenLock sync.Mutex
    sourcesSeen util.StringSet
    // the EventRecorder to use
    recorder record.EventRecorder
}

```

我们看到 podStorage 的关键属性解释如下。

- (1) pods：类型是 Map，存放每个 Pod Source 上拉过来的 Pod 数据，是 podStorage 当前保存“全量 Pod”的地方。
- (2) updates：它就是 PodConfig 里的 updates 属性的一个引用。
- (3) mode：表明 podStorage 的 Pod 事件通知模式，有以下几种。
 - ◎ PodConfigNotificationSnapshot：全量快照通知模式。
 - ◎ PodConfigNotificationSnapshotAndUpdates：全量快照+更新 Pod 通知模式（代码中创建 podStorage 实例时采用的模式）。
 - ◎ PodConfigNotificationIncremental：增量通知模式。

podStorage 实现的 Merge 接口的源码如下：

```
func (s *podStorage) Merge(source string, change interface{}) error {
    s.updateLock.Lock()
    defer s.updateLock.Unlock()
    adds, updates, deletes := s.merge(source, change)
    // deliver update notifications
    switch s.mode {
    case PodConfigNotificationSnapshotAndUpdates:
        if len(updates.Pods) > 0 {
            s.updates <- *updates
        }
        if len(deletes.Pods) > 0 || len(adds.Pods) > 0 {
            s.updates<- kubelet.PodUpdate{s.MergedState().([]*api.Pod), kubelet.
SET, source}
        }
    //省略无关的 Case 逻辑
    }
    return nil
}
```

在上述 Merge 过程中，先调用内部函数 merge，将 Pod Soucre 的 Channel 上发来的 PodUpdate 事件分解为对应的新增、更新及删除等三类 PodUpdate 事件，然后判断是否有更新事件，如果有，则直接写入汇总的 Channel 中（podStorage.updates），然后调用 MergedState 函数复制一份 podStorage 的当前全量 Pod 列表，以此产生一个全量的 PodUpdate 事件并写入汇总的 Channel 中，从而实现了多 Pod Source Channel 的“汇聚”逻辑。

分析完 Merger 过程以后，我们接下来看看是什么对象，以及如何消费这个汇总的 Channel。在上一节提到，在 kubelet 进程启动的过程中调用了 startKubelet 方法，此方法首先启动一个协程，让 kubelet 处理来自 PodSource 的 Pod Update 消息，即下面这行代码：

```
go util.Forever(func() { k.Run(podCfg.Updates()) }, 0)
```

其中，PodConfig 的 Updates()方法返回了前面我们所说的汇总 Channel 变量的一个引用，下面是 kubelet 的 Run (updates <-chan PodUpdate)方法的代码：

```
func (kl *Kubelet) Run(updates <-chan PodUpdate) {
    if kl.logServer == nil {
        kl.logServer = http.StripPrefix("/logs/",
http.FileServer(http.Dir("/var/log/")))
    }
    if kl.kubeClient == nil {
        glog.Warning("No api server defined - no node status update will be sent. ")
    }
    // Move Kubelet to a container.
    if kl.resourceContainer != "" {
        err := util.RunInResourceContainer(kl.resourceContainer)
        if err != nil {
            glog.Warningf("Failed to move Kubelet to container %q: %v", kl.
resourceContainer, err)
        }
        glog.Infof("Running in container %q", kl.resourceContainer)
    }
    if err := kl.imageManager.Start(); err != nil {
        kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed", "Failed to start
ImageManager %v", err)
        glog.Errorf("Failed to start ImageManager, images may not be garbage
collected: %v", err)
    }
    if err := kl.cadvisor.Start(); err != nil {
        kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed", "Failed to start
CAdvisor %v", err)
        glog.Errorf("Failed to start CAdvisor, system may not be properly monitored:
%v", err)
    }
    if err := kl.containerManager.Start(); err != nil {
        kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed", "Failed to start
ContainerManager %v", err)
        glog.Errorf("Failed to start ContainerManager, system may not be properly
isolated: %v", err)
    }
    if err := kl.oomWatcher.Start(kl.nodeRef); err != nil {
        kl.recorder.Eventf(kl.nodeRef, "kubeletSetupFailed", "Failed to start
OOM watcher %v", err)
        glog.Errorf("Failed to start OOM watching: %v", err)
    }
}
```

```

go util.Until(kl.updateRuntimeUp, 5*time.Second, util.NeverStop)
    // Run the system oom watcher forever.
    kl.statusManager.Start()
    kl.syncLoop(updates, kl)
}

```

上述代码首先启动了一个 HTTP File Server 来远程获取本节点的系统日志，接下来根据启动参数的设置来决定是否在指定的 Docker 容器中启动 kubelet 进程（如果成功，则将本进程转移到指定的容器中），然后分别启动 Image Manager（负责 Image GC）、cAdvisor（Docker 性能监控）、Container Manager（Container GC）、OOM Watcher（OOM 监测）、Status Manager（负责同步本节点上 Pod 的状态到 API Server 上）等组件，最后进入 syncLoop 方法中，无限循环调用下面的 syncLoopIteration 方法：

```

func (kl *Kubelet) syncLoopIteration(updates <-chan PodUpdate, handler
SyncHandler) {
    kl.syncLoopMonitor.Store(time.Now())
    if !kl.containerRuntimeUp() {
        time.Sleep(5 * time.Second)
        glog.Infof("Skipping pod synchronization, container runtime is not up. ")
        return
    }
    if !kl.doneNetworkConfigure() {
        time.Sleep(5 * time.Second)
        glog.Infof("Skipping pod synchronization, network is not configured")
        return
    }
    unsyncedPod := false
    podSyncTypes := make(map[types.UID]SyncPodType)
    select {
    case u, ok := <-updates:
        if !ok {
            glog.Errorf("Update channel is closed. Exiting the sync loop. ")
            return
        }
        kl.podManager.UpdatePods(u, podSyncTypes)
        unsyncedPod = true
        kl.syncLoopMonitor.Store(time.Now())
    case <-time.After(kl.resyncInterval):
        glog.V(4).Infof("Periodic sync")
    }
    start := time.Now()
    // If we already caught some update, try to wait for some short time
    // to possibly batch it with other incoming updates.
    for unsyncedPod {
        select {
        case u := <-updates:

```

```

        kl.podManager.UpdatePods(u, podSyncTypes)
        kl.syncLoopMonitor.Store(time.Now())
    case <-time.After(5 * time.Millisecond):
        // Break the for loop.
        unsyncedPod = false
    }
}
pods, mirrorPods := kl.podManager.GetPodsAndMirrorMap()
kl.syncLoopMonitor.Store(time.Now())
if err := handler.SyncPods(pods, podSyncTypes, mirrorPods, start); err !=
nil {
    glog.Errorf("Couldn't sync containers: %v", err)
}
kl.syncLoopMonitor.Store(time.Now())
}

```

在上述代码中，如果从 Channel 中拉取到了 PodUpdate 事件，则先调用 podManager 的 UpdatePods 方法来确定此 PodUpdate 的同步类型，并将结果放入 podSyncTypes 这个 Map 中，同时为了提升处理效率，在代码中增加了持续循环拉取 PodUpdate 数据直到 Channel 为空为止（超时判断）的一段逻辑。在方法的最后，调用 SyncHandler 接口来完成 Pod 同步的具体逻辑，从而实现了 PodUpdate 事件的高效批处理模式。

SyncHandler 在这里就是 kubelet 实例本身，它的 SyncPods 方法比较长，其主要逻辑如下。

- ◎ 将传入的全量 Pod，与 statusManager 中当前保存的 Pod 集合进行对比，删除 statusManager 中当前已经不存在的 Pod（孤儿 Pod）。
- ◎ 调用 kubelet 的 admitPods 方法以过滤掉不适合本节点创建的 Pod。此方法首先过滤掉状态为 Failed 或者 Succeeded 的 Pod；接着过滤掉不适合本节点的 Pod，比如 Host Port 冲突、Node Label 的约束不匹配及 Node 的可用资源不足等情况；最后检查磁盘的使用情况，如果磁盘的可用空间不足，则过滤掉所有 Pod。
- ◎ 对上述过滤后的 Pod 集合中的每一个 Pod 调用 podWorkers 的 UpdatePod 方法，而此方法内部创建了一个 Pod 的 workUpdate 事件并发布到该 Pod 对应的一个 Work Channel 上（podWorkers.workUpdate）。
- ◎ 对于已经删除或不存在的 Pod，通知 podWorkers 删除相关联的 Work Channel（workUpdate）。
- ◎ 对比 Node 当前运行中的 Pod 及目标 Pod 列表，“杀掉”多余的 Pod，并且调用 Docker Runtime（Docker Deamon 进程）API，重新获取当前运行中的 Pod 列表信息。
- ◎ 清理“孤儿”Pod 所遗留的 PV 和磁盘目录。

要真正理解 Pod 是怎么在 Node 上“落地”的，还要继续深入分析上述第 3 步的代码。首先我们看看对 workUpdate 这个结构体的定义：

```

type workUpdate struct {
    pod *api.Pod
    // The mirror pod of pod; nil if it does not exist.
    mirrorPod *api.Pod
    // Function to call when the update is complete.
    updateCompleteFn func()
    updateType SyncPodType
}

```

其中的属性 `pod` 是当前要操作的 Pod 对象，`mirrorPod` 则是对应的镜像 Pod，下面是对它的解释：

“对于每个来自非 API Server Pod Source 上的 Pod，kubelet 都在 API Server 上注册一个几乎“一模一样”的 Pod，这个 Pod 被称为 `mirrorPod`，这样一来，就将不同的 Pod Source 上的 Pod 都“统一”到了 kubelet 的注册表上，从而统一了 Pod 生命周期的管理流程。”

`workUpdate` 的 `updateCompleteFn` 属性是一个回调函数，`work` 完成后会执行此回调函数，在上述第 3 步中，此函数用来计算该 `work` 的调度时延指标。

对于每个要同步的 Pod，`podWorkers` 会用一个长度为 1 的 Channel 来存放其对应的 `workUpdate`，而属性 `lastUndeliveredWorkUpdate` 则存放最近一个待安排执行的 `workUpdate`，这是因为一个 Pod 的前一个 `workUpdate` 正在执行的时候，可能会有一个新的 `PodUpdate` 事件需要处理。理解了这个过程后，再来看 `podWorkers` 的定义，就不难了：

```

type podWorkers struct {
    // Protects all per worker fields.
    podLock sync.Mutex
    podUpdates map[types.UID]chan workUpdate
    isWorking map[types.UID]bool
    lastUndeliveredWorkUpdate map[types.UID]workUpdate
    runtimeCache kubecontainer.RuntimeCache
    syncPodFn syncPodFnType
    recorder record.EventRecorder
}

```

下面这个函数就是第 3 步里产生 `workUpdate` 事件并放入到 `podWorkers` 的对应 Channel 的方法的源码：

```

func (p *podWorkers) UpdatePod(pod *api.Pod, mirrorPod *api.Pod, updateComplete func()) {
    uid := pod.UID
    var podUpdates chan workUpdate
    var exists bool
    updateType := SyncPodUpdate
    p.podLock.Lock()
    defer p.podLock.Unlock()
    if podUpdates, exists = p.podUpdates[uid]; !exists {

```

```

podUpdates = make(chan workUpdate, 1)
p.podUpdates[uid] = podUpdates
updateType = SyncPodCreate
go func() {
    defer util.HandleCrash()
    p.managePodLoop(podUpdates)
}()
}
if !p.isWorking[pod.UID] {
    p.isWorking[pod.UID] = true
    podUpdates <- workUpdate{
        pod:           pod,
        mirrorPod:     mirrorPod,
        updateCompleteFn: updateComplete,
        updateType:     updateType,
    }
} else {
    p.lastUndeliveredWorkUpdate[pod.UID] = workUpdate{
        pod:           pod,
        mirrorPod:     mirrorPod,
        updateCompleteFn: updateComplete,
        updateType:     updateType,
    }
}
}
}

```

上面的代码会调用 podWorkers 的 managePodLoop 方法来处理 podUpdates 队列，这里主要是获取必要的参数，最终处理又转手交给 syncPodFn 方法去处理。下面是 managePodLoop 的源码：

```

func (p *podWorkers) managePodLoop(podUpdates <-chan workUpdate) {
    var minRuntimeCacheTime time.Time
    for newWork := range podUpdates {
        func() {
            defer p.checkForUpdates(newWork.pod.UID, newWork.updateCompleteFn)
            if err := p.runtimeCache.ForceUpdateIfOlder(minRuntimeCacheTime); err != nil {
                glog.Errorf("Error updating the container runtime cache: %v", err)
                return
            }
            pods, err := p.runtimeCache.GetPods()
            if err != nil {
                glog.Errorf("Error getting pods while syncing pod: %v", err)
                return
            }
            err = p.syncPodFn(newWork.pod, newWork.mirrorPod,
                kubecontainer.Pods(pods).FindPodByID(newWork.pod.UID), newWork.
                updateType)
            if err != nil {

```

```
    glog.Errorf("Error syncing pod %s, skipping: %v", newWork.pod.UID, err)
    p.recorder.Eventf(newWork.pod, "failedSync", "Error syncing pod, skipping: %v",
err)
    return
}
minRuntimeCacheTime = time.Now()
newWork.updateCompleteFn()
}()
}
}
```

追踪 podWorkers 的构造函数调用过程，可以发现 syncPodFn 函数其实就是 kubelet 的 syncPod 方法，这个方法的代码量有点儿多，主要逻辑如下。

(1) 根据系统配置中的权限控制，检查 Pod 是否有权在本节点运行，这些权限包括 Pod 是否有权使用 HostNetwork（还记得之前分析的代码么？由 Pod Source 类型决定）、Pod 中的容器是否被授权以特权模式启动（privileged mode）等，如果未被授权，则删除当前运行中的旧版本的 Pod 实例并返回错误信息。

(2) 创建 Pod 相关的工作目录、PV 存放目录、Plugin 插件目录，这些目录都以 Pod 的 UID 为上一级目录。

(3) 如果 Pod 有 PV 定义，则针对每个 PV 执行目录的 mount 操作。

(4) 如果是 SyncPodUpdate 类型的 Pod，则从 Docker Runtime 的 API 接口查询获取 Pod 及相关容器的最新状态信息。

(5) 如果 Pod 有 imagePullSecrets 属性，则在 API Server 上获取对应的 Secret。

(6) 调用 Container Runtime 的 API 接口方法 SyncPod，实现 Pod “真正同步”的逻辑。

(7) 如果 Pod Source 不来自 API Server，则继续处理其关联的 mirrorPod。

◎ 如果 mirrorPod 跟当前 Pod 的定义不匹配，则它会被删除。

◎ 如果 mirrorPod 还不存在（比如新创建的 Pod），则会在 API Server 上新建一个。

Kubernetes 中 Container Runtime 的默认实现是 Dockers，对应类是 dockertools.DockerManager，其源码位于 kg/kubelet/dockertools/manager.go 里，在上述 kubelet.syncPod 方法中所调用的 DockerManager 的 SyncPod 方法实现了下面的逻辑。

◎ 判断一个 Pod 实例的哪些组成部分需要重启：包括 Pod 的 infra 容器是否发生变化（如网络模式、Pod 里运行的各个容器的端口是否发生变化）；Pod 里运行的容器是否发生变化；用 Probe 检测容器的状态以确定容器是否异常等。

◎ 根据 Pod 实例重启结果的判断，如果需要重启 Pod 的 infra 容器，则先 Kill Pod 然后启

动 Pod 的 infra 容器，设定好网络，最后启动 Pod 里的所有 Container；否则就先 Kill 那些需要重启的 Container，然后重新启动它们。注意，如果是新创建的 Pod，则因为找不到 Node 上对应的 Pod 的 infra 容器，所以会被当作重启 Pod 的 infra 容器的逻辑来实现创建过程。

DockerManager 创建 Pod 的 infra 容器的逻辑在 `createPodInfraContainer` 方法里，大体逻辑如下。

- ◎ 如果 Pod 的网络不是 HostNetwork 模式，则搜集 Pod 所有容器的 Port 作为 infra 容器所要暴露的 Port 列表。
- ◎ 如果 infra 容器的 Image 目前不存在，则尝试拉取 Image。
- ◎ 创建 infra 的 Container 对象并且启动 `runContainerInPod` 方法。
- ◎ 如果容器定义有 Lifecycle，并且 PostStart 回调方法被设置了，就会触发此方法的调用，如果调用失败则 Kill 容器并返回。
- ◎ 创建一个软连接文件指向容器的日志文件，此软连接文件名包括 Pod 的名称、容器的名称及容器的 ID，这样的目的是让 ElasticSearch 这样的搜索技术容易索引和定位 Pod 日志。
- ◎ 如果此容器是 Pod infra 容器，则设置其 OOM 参数低于标准值，使得它比其他容器具备更强的“抗灾”能力。
- ◎ 修改 Docker 生成的容器的 resolv.conf 文件，增加 ndots 参数并默认设置为 5，这是因为 Kubernetes 默认假设的域名分割长度是 5，例如 _dns._udp.kube-dns.default.svc。

上述逻辑中所调用的 `runContainerInPod` 是 DockerManager 的核心方法之一，不管是创建 Pod 的 infra 容器还是 Pod 里的其他容器，都会通过此方法使得容器被创建和运行。以下是其主要逻辑。

- ◎ 生成 Container 必要的环境变量和参数，比如 ENV 环境变量、Volume Mounts 信息、端口映射信息、DNS 服务器信息、容器的日志目录、parent cgGroup 等。
- ◎ 调用 `runContainer` 方法完成 Docker Container 实例的创建过程，简单地说，就是完成 Docker create container 命令行所需的各种参数的构造过程，并通过程序来调用执行。
- ◎ 构造 HostConfig 对象，主要参数有目录映射、端口映射等、cgGroup 的设定等，简单地说，就是完成了 Docker start container 命令行所需的必要参数的构造过程，并通过程序来调用执行。

在上述逻辑中，`runContainer` 与 `startContainer` 的具体实现都是靠 DockerManager 中的 `dockerClient` 对象完成的，它实现了 DockerInterface 接口，`dockerClient` 的创建过程在 `pkg/kubelet/dockertools/docker.go` 里，下面是这段代码：

```
func ConnectToDockerOrDie(dockerEndpoint string) DockerInterface {
```

```

    if dockerEndpoint == "fake:// " {
        return &FakeDockerClient{
            VersionInfo: docker.Env{"ApiVersion=1.18"},
        }
    }
    client, err := docker.NewClient(getDockerEndpoint(dockerEndpoint))
    if err != nil {
        glog.Fatalf("Couldn't connect to docker: %v", err)
    }
    return client
}

```

这里的 dockerEndpoint 是本节点上的 Docker Deamon 进程的访问地址，默认是 unix:///var/run/docker.sock，在上述代码中使用了来自开源项目 <https://github.com/fsouza/go-dockerclient> 提供的 Docker Client，它也是 Go 语言实现的一个用 HTTP 访问 Docker Deamon 提供的标准 API 的客户端框架。

我们来看看 dockerClient 创建容器的具体代码（CreateContainer）：

```

func (c *Client) CreateContainer(opts CreateContainerOptions) (*Container, error) {
    path := "/containers/create? " + queryString(opts)
    body, status, err := c.do(
        "POST",
        path,
        doOptions{
            data: struct {
                *Config
                HostConfig *HostConfig `json: "HostConfig,omitempty" yaml:
"HostConfig,omitempty"`
            }{
                opts.Config,
                opts.HostConfig,
            },
        },
    )
    if status == http.StatusNotFound {
        return nil, ErrNoSuchImage
    }
    if err != nil {
        return nil, err
    }
    var container Container
    err = json.Unmarshal(body, &container)
    if err != nil {
        return nil, err
    }
    container.Name = opts.Name
}

```

```

    return &container, nil
}

```

上述代码其实就是通过调用标准的 Docker Rest API 来实现功能的，我们进入 docker.Client 的 do 方法里可以看到更多详情，例如输入参数转换为 JSON 格式的数据、DockerAPI 版本检查及异常处理等逻辑，最有趣的是：在 dockerEndpoint 是 unix 套接字的情况下，会先建立套接字连接，然后在这个连接上创建 HTTP 连接。

至此，我们分析了 kubelet 创建和同步 Pod 实例的整个流程，简单总结如下。

- ◎ 汇总：先将多个 Pod Source 上过来的 PodUpdate 事件汇聚到一个总的 Channel 上去。
- ◎ 初审：分析并过滤掉不符合本节点的 PodUpdate 事件，对满足条件的 PodUpdate 则生成一个 workUpdate 事件，交给 podWorkers 处理。
- ◎ 接待：podWorkers 对每个 Pod 的 workUpdate 事件排队，并且负责更新 Cache 中的 Pod 状态，而把具体的任务转给 kubelet 去处理（syncPod 方法）。
- ◎ 终审：kubelet 对符合条件的 Pod 进一步审查，例如检查 Pod 是否有权在本节点运行，对符合审查的 Pod 开始着手准备工作，包括目录创建、PV 创建、Image 获取、处理 Mirror Pod 问题等，然后把“皮球”踢给了 DockerManager。
- ◎ 落地：任务抵达 DockerManager 之后，DockerManager 尽心尽责地分析每个 Pod 的情况，以决定这个 Pod 究竟是新建、完全重启还是部分更新的。给出分析结果以后，剩下的就是 dockerClient 的工作了。

好复杂的设计！原来非业务流程的代码理解起来也会如此折磨人，真心不知道谷歌当初是怎么设计和实现它的，目测国内 P8 水平的一帮大牛们天天加班到 9 点钟，也难以交付这样的 Code。

在继续下面的分析之前，留一个小小的思考给聪明的读者：Pod Source 上发来的 Pod 删除的事件，是在哪里处理的？

接下来我们继续分析 kubelet 进程的另外一个重要功能是如何实现的，即定期同步 Pod 状态信息到 API Server 上。先来看看 Pod 状态的数据结构定义：

```

type PodStatus struct {
    Phase      PodPhase      `json: "phase,omitempty"`
    Conditions []PodCondition `json: "conditions,omitempty"`
    Message string `json: "message,omitempty"`
    Reason string `json: "reason,omitempty"`
    HostIP string `json: "hostIP,omitempty"`
    PodIP string `json: "podIP,omitempty"`
    StartTime *util.Time `json: "startTime,omitempty"`
    ContainerStatuses []ContainerStatus
}
// PodStatusResult is a wrapper for PodStatus returned by kubelet that can be

```

```
encode/decoded
type PodStatusResult struct {
    TypeMeta `json: ",inline"`
    ObjectMeta `json: "metadata,omitempty"`
    Status PodStatus `json: "status,omitempty"`
}
```

Pod 的状态 (Phase) 有 5 种：运行中 (PodRunning)、等待中 (PodPending)、正常终止 (PodSucceeded)、异常停止 (PodFailed) 及未知状态 (PodUnknown)，最后一种状态很可能是由于 Pod 所在主机的通信问题导致的。从上面的定义可以看到 Pod 的状态同时包括它里面运行的 Container 的状态，另外给出了导致当前状态的原因说明、Pod 的启动时间等信息。PodStatusResult 则是 Kubelete API Server 提供的 Pod Status API 接口中用到的 Wrapper 类。

通过之前的代码研读，我们发现在 Kubernetes 中大量使用了 Channel 和协程机制来完成数据的高效传递和处理工作，在 kubelet 中更是大量使用了这一机制，实现 Pod Status 上报的 kubelet.statusManager 也是如此，它用一个 Map (podStatuses) 保存了当前 kubelet 中所有 Pod 实例的当前状态，并且声明了一个 Channel (podStatusChannel) 来存放 Pod 状态同步的更新请求 (podStatuses)，Pod 在本地实例化和同步的过程中会引发 Pod 状态的变化，这些变化被封装为 podStatusSyncRequest 放入 Channel 中，然后被异步上报到 API Server，这就是 statusManager 的运行机制。

下面是 statusManager 的 SetPodStatus 方法，先比较缓存的状态信息，如果状态发生变化，则触发 Pod 状态，生成 podStatusSyncRequest 并放到队列中等待上报：

```
func (s *statusManager) SetPodStatus(pod *api.Pod, status api.PodStatus) {
    podFullName := kubecontainer.GetPodFullName(pod)
    s.podStatusesLock.Lock()
    defer s.podStatusesLock.Unlock()
    oldStatus, found := s.podStatuses[podFullName]
    // ensure that the start time does not change across updates.
    if found && oldStatus.StartTime != nil {
        status.StartTime = oldStatus.StartTime
    }
    if status.StartTime.IsZero() {
        if pod.Status.StartTime.IsZero() {
            // the pod did not have a previously recorded value so set to now
            now := util.Now()
            status.StartTime = &now
        } else {
            status.StartTime = pod.Status.StartTime
        }
    }
    if !found || !isStatusEqual(&oldStatus, &status) {
        s.podStatuses[podFullName] = status
    }
}
```

```

        s.podStatusChannel <- podStatusSyncRequest{pod, status}
    } else {
        glog.V(3).Infof("Ignoring same status for pod %q, status: %v", kubeletUtil.
FormatPodName(pod), status)
    }
}
}

```

下面是在 Pod 实例化的过程中, kubelet 过滤掉不合适本节点 Pod 所调用的上述方法的代码, 类似的调用还有不少:

```

func (kl *Kubelet) handleNotFittingPods(pods []*api.Pod) []*api.Pod {
    fitting, notFitting := checkHostPortConflicts(pods)
    for _, pod := range notFitting {
        reason := "HostPortConflict"
        kl.recorder.Eventf(pod, reason, "Cannot start the pod due to host port
conflict. ")
        kl.statusManager.SetPodStatus(pod, api.PodStatus{
            Phase:  api.PodFailed,
            Reason: reason,
            Message: "Pod cannot be started due to host port conflict"})
    }
    fitting, notFitting = kl.checkNodeSelectorMatching(fitting)
    for _, pod := range notFitting {
        reason := "NodeSelectorMismatching"
        kl.recorder.Eventf(pod, reason, "Cannot start the pod due to node selector
mismatch. ")
        kl.statusManager.SetPodStatus(pod, api.PodStatus{
            Phase:  api.PodFailed,
            Reason: reason,
            Message: "Pod cannot be started due to node selector mismatch"})
    }
    fitting, notFitting = kl.checkCapacityExceeded(fitting)
    for _, pod := range notFitting {
        reason := "CapacityExceeded"
        kl.recorder.Eventf(pod, reason, "Cannot start the pod due to exceeded
capacity. ")
        kl.statusManager.SetPodStatus(pod, api.PodStatus{
            Phase:  api.PodFailed,
            Reason: reason,
            Message: "Pod cannot be started due to exceeded capacity"})
    }
    return fitting
}

```

最后, 我们看看 statusManager 是怎么把 Channel 的数据上报到 API Server 上的, 这是通过 Start 方法开启一个协程无限循环执行 syncBatch 方法来实现的, 下面是 syncBatch 的代码:

```
func (s *statusManager) syncBatch() error {
```

```

syncRequest := <-s.podStatusChannel
pod := syncRequest.pod
podFullName := kubecontainer.GetPodFullName(pod)
status := syncRequest.status

var err error
statusPod := &api.Pod{
    ObjectMeta: pod.ObjectMeta,
}
statusPod, err = s.kubeClient.Pods(statusPod.Namespace).Get(statusPod.Name)
if err == nil {
    statusPod.Status = status
    _, err = s.kubeClient.Pods(pod.Namespace).UpdateStatus(statusPod)
    // TODO: handle conflict as a retry, make that easier too.
    if err == nil {
        glog.V(3).Infof("Status for pod %q updated successfully", kubeletUtil.
FormatPodName(pod))
        return nil
    }
}
go s.DeletePodStatus(podFullName)
return fmt.Errorf("error updating status for pod %q: %v",
kubeletUtil.FormatPodName(pod), err)
}

```

这段代码首先从 Channel 中拉取一个 syncRequest，然后调用 API Server 接口来获取最新的 Pod 信息，如果成功，则继续调用 API Server 的 UpdateStatus 接口更新 Pod 状态，如果调用失败则删除缓存的 Pod 状态，这将触发 kubelet 重新计算 Pod 状态并再次尝试更新。

说完了 Pod 流程，我们接下来再一起深入分析 Kubernetes 中的容器探针（Probe）的实现机制。我们知道，容器正常不代表里面运行的业务进程能正常工作，比如程序还没初始化好，或者配置文件错误导致无法正常服务，还有诸如数据库连接爆满导致服务异常等各种意外情况都有可能发生，面对这类问题，cAdvisor 就束手无策了，所以 kubelet 引入了容器探针技术，容器探针按照作用划分为以下两种。

- ① **ReadinessProbe**: 用来探测容器中的用户服务进程是否处于“可服务状态”，此探针不会导致容器被停止或重启，而是导致此容器上的服务被标识为不可用，Kubernetes 不会发送请求到不可用的容器上，直到它们可用为止。
- ② **LivenessProbe**: 用来探测容器服务是否处于“存活状态”，如果服务当前被检测为 Dead，则会导致容器重启事件发生。

下面是探针相关的结构定义：

```
type Probe struct {
```

```

Handler
InitialDelaySeconds int64
TimeoutSeconds int64
}
type Handler struct {
    // One and only one of the following should be specified.
    Exec *ExecAction
    HTTPGet *HTTPGetAction
    TCPSocket *TCPSocketAction
}

```

从上面的定义来看，探针可以通过执行容器中的一个命令、发起一个指向容器内部的 HTTP Get 请求或者 TCP 连接来确定容器内部是否正常工作。

上面的代码属于 API 包中的一部分，只是用来描述和存储容器上的探针定义，而真正的探针实现代码则位于 `pkg/kubelet/prober/prober.go` 里，下面是对 `prober.Probe` 的定义：

```

type Prober interface {
    Probe(pod *api.Pod, status api.PodStatus, container api.Container, containerID
string, createdAt int64) (probe.Result, error)
}

```

上述接口方法表示对一个 `Container` 发起探测并返回其结果。`prober.Probe` 的实现类为 `prober.prober`，其结构定义如下：

```

type prober struct {
    exec  execprobe.ExecProber
    http  httpprobe.HTTPProber
    tcp   tcprobe.TCPProber
    runner kubecontainer.ContainerCommandRunner
    readinessManager *kubecontainer.ReadinessManager
    refManager      *kubecontainer.RefManager
    recorder        record.EventRecorder
}

```

其中 `exec`、`http`、`tcp` 三个变量分别对应三种探测类型的“探头”，它们已经各自实现了相应的逻辑。比如下面这段代码是 HTTP 探头的核心逻辑，即连接一个 URL 发起 GET 请求：

```

func DoHTTPProbe(url *url.URL, client HTTPGetInterface) (probe.Result, string,
error) {
    res, err := client.Get(url.String())
    if err != nil {
        // Convert errors into failures to catch timeouts.
        return probe.Failure, err.Error(), nil
    }
    defer res.Body.Close()
    b, err := ioutil.ReadAll(res.Body)
    if err != nil {
        return probe.Failure, "", err
    }
}

```

```

    }
    body := string(b)
    if res.StatusCode >= http.StatusOK && res.StatusCode < http.StatusBadRequest {
        glog.V(4).Infof("Probe succeeded for %s, Response: %v", url.String(), *res)
    } else {
        glog.V(4).Infof("Probe failed for %s, Response: %v", url.String(), *res)
    }
}

```

prober.prober 中的 runner 则是 exec 探头的执行器，因为后者需要在被检测的容器中执行一个 cmd 命令：

```

func (p *prober) newExecInContainer(pod *api.Pod, container api.Container,
containerID string, cmd []string) exec.Cmd {
    return execInContainer(func() ([]byte, error) {
        return p.runner.RunInContainer(containerID, cmd)
    })
}

```

实际上 p.runner 就是之前我们分析过的 DockerManager，下面是 RunInContainer 的源码：

```

func (dm *DockerManager) RunInContainer(containerID string, cmd []string)
([]byte, error) {
    // If native exec support does not exist in the local docker daemon use nsinit.
    useNativeExec, err := dm.nativeExecSupportExists()
    if err != nil {
        return nil, err
    }
    if !useNativeExec {
        glog.V(2).Infof("Using nsinit to run the command %v inside container %s",
            cmd, containerID)
        return dm.runInContainerUsingNsinit(containerID, cmd)
    }
    glog.V(2).Infof("Using docker native exec to run cmd %v inside container %s",
        cmd, containerID)
    createOpts := docker.CreateExecOptions{
        Container:    containerID,
        Cmd:          cmd,
        AttachStdin:  false,
        AttachStdout: true,
        AttachStderr: true,
        Tty:          false,
    }
    execObj, err := dm.client.CreateExec(createOpts)
    if err != nil {
        return nil, fmt.Errorf("failed to run in container - Exec setup failed")
    }
}

```

```

- %v", err)
}
var buf bytes.Buffer
startOpts := docker.StartExecOptions{
    Detach:     false,
    Tty:        false,
    OutputStream: &buf,
    ErrorStream: &buf,
    RawTerminal: false,
}
err = dm.client.StartExec(execObj.ID, startOpts)
if err != nil {
    glog.V(2).Infof("StartExec With error: %v", err)
    return nil, err
}
ticker := time.NewTicker(2 * time.Second)
defer ticker.Stop()
for {
    inspect, err2 := dm.client.InspectExec(execObj.ID)
    if err2 != nil {
        glog.V(2).Infof("InspectExec %s failed with error: %+v", execObj.
ID, err2)
        return buf.Bytes(), err2
    }
    if !inspect.Running {
        if inspect.ExitCode != 0 {
            glog.V(2).Infof("InspectExec %s exit with result %+v", execObj.
ID, inspect)
            err = &dockerExitError{inspect}
        }
        break
    }
    <-ticker.C
}
return buf.Bytes(), err
}

```

Docker 自 1.3 版本开始支持使用 Exec 指令（以及 API 调用）在容器内执行一个命令，我们看看上述过程中使用的 `dm.client.CreateExec` 方法是如何实现的：

```

func (c *Client) CreateExec(opts CreateExecOptions) (*Exec, error) {
    path := fmt.Sprintf("/containers/%s/exec", opts.Container)
    body, status, err := c.do("POST", path, doOptions{data: opts})
    if status == http.StatusNotFound {
        return nil, &NoSuchContainer{ID: opts.Container}
    }
}

```

```

    if err != nil {
        return nil, err
    }
    var exec Exec
    err = json.Unmarshal(body, &exec)
    if err != nil {
        return nil, err
    }
    return &exec, nil
}

```

我们看到，这是标准的 Docker API 的调用方式，跟之前看到的创建容器的调用代码很相似。现在我们再回头看看 prober.prober 是怎么执行 ReadinessProbe/ LivenessProbe 的检测逻辑的：

```

func (pb *prober) Probe(pod *api.Pod, status api.PodStatus, container api.Container, containerID string, createdAt int64) (probe.Result, error) {
    pb.probeReadiness(pod, status, container, containerID, createdAt)
    return pb.probeLiveness(pod, status, container, containerID, createdAt)
}

```

这段代码先调用容器的 ReadinessProbe 进行检测，并且在 readinessManager 组件中记录容器的 Readiness 状态，随后调用容器的 LivenessProbe 进行检测，并返回容器的状态，在检测过程中如果发现状态为失败或者异常状态，则会连续检测 3 次：

```

func (pb *prober) runProbeWithRetries(p *api.Probe, pod *api.Pod, status api.PodStatus, container api.Container, containerID string, retries int) (probe.Result, string, error) {
    var err error
    var result probe.Result
    var output string
    for i := 0; i < retries; i++ {
        result, output, err = pb.runProbe(p, pod, status, container, containerID)
        if result == probe.Success {
            return probe.Success, output, nil
        }
    }
    return result, output, err
}

```

比较意外的是 prober.prober 探针检测容器状态的方法目前只在一处被调用到，位于方法 DockerManager.computePodContainerChanges 里：

```

result, err := dm.prober.Probe(pod, podStatus, container, string(c.ID), c.Created)
if err != nil {
    // TODO(vmarmol): examine this logic.
    glog.V(2).Infof("probe no-error: %q", container.Name)
    containersToKeep[containerID] = index
}

```

```

        continue
    }
    if result == probe.Success {
        glog.V(4).Infof("probe success: %q", container.Name)
        containersToKeep[containerID] = index
        continue
    }
    glog.Infof("pod %q container %q is unhealthy (probe result: %v), it will
be killed and re-created. ", podFullName, container.Name, result)
    containersToStart[index] = empty{}
}

```

只有没有发生任何变化的 Pod 才会执行一次探针检测，若检测状态为失败，则会导致重启事件发生。

本节最后，我们再来简单分析下 kubelet 中的 Kubelet Server 的实现机制，下面是 kubelet 进程启动过程中启动 Kubelet Server 的源码入口：

```

// start the kubelet server
if kc.EnableServer {
    go util.Forever(func() {
        k.ListenAndServe(net.IP(kc.Address), kc.Port, kc.TLSOptions, kc.
EnableDebuggingHandlers)
    }, 0)
}

```

在上述代码调用的过程中，创建了一个类型为 kubelet.Server 的 HTTP Server 并在本地监听：

```

handler := NewServer(host, enableDebuggingHandlers)
s := &http.Server{
    Addr:             net.JoinHostPort(address.String(), strconv.FormatUint
(uint64(port), 10)),
    Handler:          &handler,
    ReadTimeout:      5 * time.Minute,
    WriteTimeout:     5 * time.Minute,
    MaxHeaderBytes:   1 << 20,
}
if tlsOptions != nil {
    s.TLSConfig = tlsOptions.Config
    glog.Fatal(s.ListenAndServeTLS(tlsOptions.CertFile, tlsOptions.KeyFile))
} else {
    glog.Fatal(s.ListenAndServe())
}

```

在 kubelet.Server 的构造函数里加载如下 HTTP Handler：

```

func (s *Server) InstallDefaultHandlers() {
    healthz.InstallHandler(s.mux,
        healthz.PingHealthz,

```

```

        healthz.NamedCheck("docker", s.dockerHealthCheck),
        healthz.NamedCheck("hostname", s.hostnameHealthCheck),
        healthz.NamedCheck("syncloop", s.syncLoopHealthCheck),
    )
    s mux.HandleFunc("/pods", s.handlePods)
    s mux.HandleFunc("/stats/", s.handleStats)
    s mux.HandleFunc("/spec/", s.handleSpec)
}

```

上述 Handler 分为两组：首先是健康检查，包括 kubelet 进程自身的心跳检查、Docker 进程的健康检查、kubelet 所在主机名检测、Pod 同步的健康检查等；然后是获取当前节点上运行期信息的接口，例如获取当前节点上的 Pod 列表、统计信息等。下面是 hostnameHealthCheck 的实现逻辑，它检查 Pod 两次同步之间的时延，而这个时延则在之前提到的 kubelet 的 syncLoopIteration 方法中进行更新：

```

func (s *Server) syncLoopHealthCheck(req *http.Request) error {
    duration := s.host.ResyncInterval() * 2
    minDuration := time.Minute * 5
    if duration < minDuration {
        duration = minDuration
    }
    enterLoopTime := s.host.LatestLoopEntryTime()
    if !enterLoopTime.IsZero() && time.Now().After(enterLoopTime.Add(duration)) {
        return fmt.Errorf("Sync Loop took longer than expected. ")
    }
    return nil
}

```

handlePods 的 API 则从 kubelet 中获取当前“绑定”到本节点的所有 Pod 的信息并返回：

```

func (s *Server) handlePods(w http.ResponseWriter, req *http.Request) {
    pods := s.host.GetPods()
    data, err := encodePods(pods)
    if err != nil {
        s.error(w, err)
        return
    }
    w.Header().Add("Content-type", "application/json")
    w.Write(data)
}

```

如果 kubelet 运行在 Debug 模式，则加载更多的 HTTP Handler：

```

func (s *Server) InstallDebuggingHandlers() {
    s mux.HandleFunc("/run/", s.handleRun)
    s mux.HandleFunc("/exec/", s.handleExec)
    s mux.HandleFunc("/portForward/", s.handlePortForward)
}

```

```

    s mux.HandleFunc("/logs/", s.handleLogs)
    s mux.HandleFunc("/containerLogs/", s.handleContainerLogs)
    s mux.Handle("/metrics", prometheus.Handler())
    // The /runningpods endpoint is used for testing only.
    s mux.HandleFunc("/runningpods", s.handleRunningPods)

    s mux.HandleFunc("/debug/pprof/", pprof.Index)
    s mux.HandleFunc("/debug/pprof/profile", pprof.Profile)
    s mux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
}

```

这些 HTTP Handler 的实现并不复杂，所以在这里就不再一一介绍了。

6.5.3 设计总结

在研读 kubelet 源码的过程中，你经常会有“山穷水尽疑无路，柳暗花明又一村”的感觉，是因为在它的设计中大量运用了 Channel 这种异步消息机制，加之为了测试的方便，又将很多重要的处理函数做成接口类，只有找到并分析这些接口的具体实现类，才能明白整个流程。这对于习惯了面向对象语言的程序员而言，有一种一夜回到解放前的感觉。

因为 kubelet 的功能比较多，所以我们在此仅以 Pod 同步的主流程为例，进行一个设计总结，图 6.8 是 kubelet 主流程相关的设计示意图，为了更加清晰地展示整个流程，我们特意将 kubelet Kernel、Docker System 与其他部分分离开来，并且省略了部分非核心对象和数据结构。

首先，config.PodConfig 创建一个或多个 Pod Source，在默认情况下创建的是 API source，它并没有创建新的数据结构，而是使用之前介绍的 cache.Reflector 结合 cache.UndeltaStore，从 Kubernetes API Server 上拉取 Pod 数据放入内部的 Channel 上，而内部的 Channel 收到 Pod 数据后会调用 podStorage 的 Merge 方法实现多个 Channel 数据的合并，产生 kubelet.PodUpdate 消息并写入 PodConfig 的汇总 Channel 上，随后 PodUpdate 消息进入 kubelet Kernel 中进行下一步处理。

kubelet.kubelet 的 syncLoop 方法监听 PodConfig 的汇总 Channel，过滤掉不合适的 PodUpdate 并把符合条件的放入 SyncPods 方法中，最终为每个符合条件的 Pod 产生一个 kubelet.workUpdate 事件并放入 podWorkers 的内部工作队列上，随后调用 podWorkers 的 managePodLoop 方法进行处理。podWorkers 在处理流程中调用了 DockerManager 的 SyncPod 方法，由此 DockerManager 接班，在进行了必要的 Pod 周边操作后，对于需要重启或者更新的容器，DockerManager 则交给 docker.Client 对象去执行具体的操作，后者通过调用 Dockers Engine 的 API Service 来实现具体功能。

在 Pod 同步的过程中会产生 Pod 状态的变更和同步问题，这些是交由 kubelet.statusManager 实现的，它在内部也采用了 Channel 的设计方式。

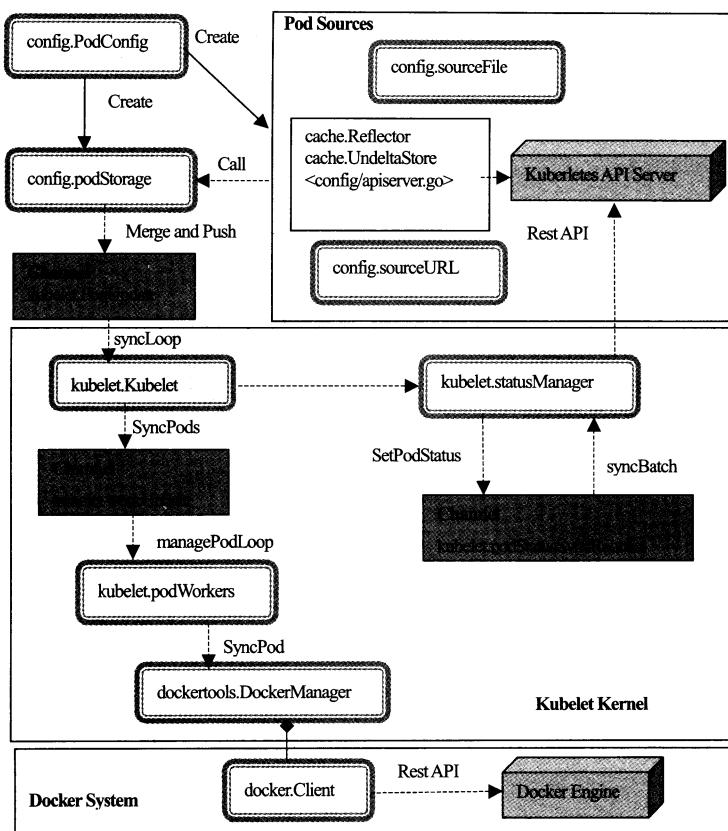


图 6.8 kubelet 主流程相关的设计示意图

6.6 kube-proxy 进程源码分析

`kube-proxy` 是运行在 Minion 节点上的另外一个重要守护进程，你可以把它当作一个 HAProxy，它充当了 Kubernetes 中 Service 的负载均衡器和服务代理的角色。下面我们分别对其启动过程、关键代码分析及设计总结等方面进行深入分析和讲解。

6.6.1 进程启动过程

`kube-proxy` 进程的入口类源码位置如下：

[github.com/GoogleCloudPlatform/kubernetes/cmd/kube-proxy/proxy.go](https://github.com/GoogleCloudPlatform/kubernetes/blob/master/cmd/kube-proxy/proxy.go)

入口 main() 函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    s := app.NewProxyServer()
    s.AddFlags(pflag.CommandLine)

    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()

    verflag.PrintAndExitIfRequested()

    if err := s.Run(pflag.CommandLine.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

上述代码构造了一个 ProxyServer，然后调用它的 Run 方法启动运行。首先我们看看 NewProxyServer 的代码：

```
func NewProxyServer() *ProxyServer {
    return &ProxyServer{
        BindAddress:      util.IP(net.ParseIP("0.0.0.0")),
        HealthzPort:     10249,
        HealthzBindAddress: util.IP(net.ParseIP("127.0.0.1")),
        OOMScoreAdj:     -899,
        ResourceContainer: "/kube-proxy",
    }
}
```

在上述代码中，ProxyServer 绑定本地所有 IP (0.0.0.0) 对外提供代理服务，而提供健康检查的 HTTP Server 则默认绑定本地的回环 IP，说明后者仅用于在本节点上访问，如果需要开发管理系统进行远程管理，则可以设置参数 healthz-bind-address 为 0.0.0.0 来达到目的。另外，从代码中看，ProxyServer 还有一个重要属性可以调整：PortRange (对应命令行参数为 proxy-port-range)，它用来限定 ProxyServer 使用哪些本地端口作为代理端口，默认是随机选择。

ProxyServer 的 Run 方法流程如下。

- ◎ 设置本进程的 OOM 参数 OOMScoreAdj，保证系统 OOM 时，kube-proxy 不会首先被系统删除，这是因为 kube-proxy 与 kubelet 进程一样，比节点上的 Pod 进程更重要。
- ◎ 让自己的进程运行在指定的 Linux Container 中，这个 Container 的名字来自 ProxyServer.ResourceContainer，如上所述，默认为 /kube-proxy，比较重要的一点是这个 Container 具备所有设备的访问权。

- ◎ 创建 ServiceConfig 与 EndpointsConfig，它们与之前 kubelet 中的 PodConfig 的作用和实现机制有点像，分别负责监听和拉取 API Server 上 Service 与 Service Endpoints 的信息，并通知给注册到它们上的 Listener 接口进行处理。
- ◎ 创建一个 round-robin 轮询机制的 load balancer (LoadBalancerRR)，它用来实现 Service 的负载均衡转发逻辑，它也是前面创建的 EndpointsConfig 的一个 Listener。
- ◎ 创建一个 Proxier，它负责建立和维护 Service 的本地代理 Socket，它也是前面创建的 ServiceConfig 的一个 Listener。
- ◎ 创建一个 config.SourceAPI，并启动两个协程，通过 Kubernetes Client 来拉取 Kubernetes API Server 上的 Service 与 Endpoint 数据，然后分别写入之前定义的 ServiceConfig 与 EndpointsConfig 的 Channel 上，从而触发整个流程的驱动。
- ◎ 本地绑定健康检查的 HTTP Server 提供服务。
- ◎ 进入 Proxier 的 SyncLoop 方法里，该方法周期性地检查 Iptables 是否设置正常、服务的 Portal 是否正常开启，以及清除 load balancer 上的过期会话。

从启动流程看，kube-proxy 进程的参数比较少，它做的事情也是比较单一的，没有 kubelet 进程那么复杂，在下一节我们会深入分析其关键代码。

6.6.2 关键代码分析

从上一节 kube-proxy 的启动流程来看，它跟 kubelet 有相似的地方，即都会从 Kubernetes API Server 拉取相关的资源数据并在本地节点上完成“深加工”，其拉取资源的做法，第一眼看上去与 kubelet 相似，但实际上有稍微不同的实现思路，这说明作者另有其人。

由于 ServiceConfig 与 EndpointsConfig 实现机制是完全一样的，只不过拉取的资源不同，所以我们这里仅对前者做深入分析。首先从 ServiceConfig 结构体开始：

```
type ServiceConfig struct {
    mux      *config.Mux
    bcaster *config.Broadcaster
    store   *serviceStore
}
```

ServiceConfig 也使用了 mux(config.Mux)，它是一个多 Channel 的多路合并器，之前 kubelet 的 PodConfig 也用到了它。下面是 ServiceConfig 的构造函数：

```
func NewServiceConfig() *ServiceConfig {
    updates := make(chan struct{})
    store := &serviceStore{updates: updates, services:
        make(map[string]map[types.NamespacedName]api.Service)}
}
```

```

    mux := config.NewMux(store)
    bcaster := config.NewBroadcaster()
    go watchForUpdates(bcaster, store, updates)
    return &ServiceConfig{mux, bcaster, store}
}

```

从上述代码来看，store 是 serviceStore 的一个实例。它作为 config.Mux 的 Merge 接口的实现，负责处理 config.Mux 的 Channel 上收到的 ServiceUpdate 消息并更新 store 的内部变量 services，后者是一个 Map，存放了最新同步到本地的 api.Service 资源，是 Service 的全量数据。下面是 Merge 方法的逻辑：

```

func (s *serviceStore) Merge(source string, change interface{}) error {
    s.serviceLock.Lock()
    services := s.services[source]
    if services == nil {
        services = make(map[types.NamespacedName]api.Service)
    }
    update := change.(ServiceUpdate)
    switch update.Op {
    case ADD:
        glog.V(4).Infof("Adding new service from source %s : %+v", source, update.Services)
        for _, value := range update.Services {
            name := types.NamespacedName{value.Namespace, value.Name}
            services[name] = value
        }
    case REMOVE:
        glog.V(4).Infof("Removing a service %+v", update)
        for _, value := range update.Services {
            name := types.NamespacedName{value.Namespace, value.Name}
            delete(services, name)
        }
    case SET:
        glog.V(4).Infof("Setting services %+v", update)
        // Clear the old map entries by just creating a new map
        services = make(map[types.NamespacedName]api.Service)
        for _, value := range update.Services {
            name := types.NamespacedName{value.Namespace, value.Name}
            services[name] = value
        }
    default:
        glog.V(4).Infof("Received invalid update type: %v", update)
    }
    s.services[source] = services
    s.serviceLock.Unlock()
    if s.updates != nil {
        s.updates <- struct{}{}
    }
}

```

```
    }
    return nil
}
```

serviceStore 同时是 config.Accessor 接口的一个实现，MergedState 接口方法返回之前 Merge 最新的 Service 全量数据。

```
func (s *serviceStore) MergedState() interface{} {
    s.serviceLock.RLock()
    defer s.serviceLock.RUnlock()
    services := make([]api.Service, 0)
    for _, sourceServices := range s.services {
        for _, value := range sourceServices {
            services = append(services, value)
        }
    }
    return services
}
```

上述方法在哪里被用到了呢？就在之前提到的 NewServiceConfig 方法里：

```
go watchForUpdates(bcaster, store, updates)
```

一个协程监听 serviceStore 的 updates(Channel)，在收到事件以后就调用上述 MergedState 方法，将当前最新的 Service 数组通知注册到 bcaster 上的所有 Listener 进行处理。下面分别给出了 watchForUpdates 及 Broadcaster 的 Notify 方法的源码：

```
func watchForUpdates(bcaster *config.Broadcaster, accessor config.Accessor,
updates <-chan struct{}{}) {
    for true {
        <-updates
        bcaster.Notify(accessor.MergedState())
    }
}
func (b *Broadcaster) Notify(instance interface{}) {
    b.listenerLock.RLock()
    listeners := b.listeners
    b.listenerLock.RUnlock()
    for _, listener := range listeners {
        listener.OnUpdate(instance)
    }
}
```

上述逻辑的精巧设计之处在于，当 ServiceConfig 完成 Merge 调用后，为了及时通知 Listener 进行处理，就产生一个“空事件”并写入 updates 这个 Channel 中，另外监听此 Channel 的协程就及时得到通知，触发 Listener 的回调动作。ServiceConfig 这里注册的 Listener 是 proxy.Proxyer 对象，我们以后会继续分析它的回调函数 OnUpdate 是如何使用 Service 数据的。

接下来，我们看看 ServiceUpdate 事件是怎么生成并传递到 ServiceConfig 的 Channel 上的。在 kube-proxy 启动流程中有调用 config.NewSourceAPI 函数，其内部生成了一个 servicesReflector 对象：

```
type servicesReflector struct {
    watcher      ServicesWatcher
    services     chan<- ServiceUpdate
    resourceVersion string
    waitDuration   time.Duration
    reconnectDuration time.Duration
}
```

其中 services 这个 Channel 是用来写入 ServiceUpdate 事件的，它是 ServiceConfig 的 Channel (source string) 方法所创建并返回的 Channel，它写入数据后就会被一个协程立即转发到 ServiceConfig 的 Channel 里。下面这段代码完整地揭示了上述逻辑：

```
func (c *ServiceConfig) Channel(source string) chan ServiceUpdate {
    ch := c.mux.Channel(source)
    serviceCh := make(chan ServiceUpdate)
    go func() {
        for update := range serviceCh {
            ch <- update
        }
        close(ch)
    }()
    return serviceCh
}
```

servicesReflector 中的 watcher 用来从 API Server 上拉取 Service 数据，它是 client.Services (api.NamespaceAll) 返回的 client.ServiceInterface 实例对象的一个引用，属于标准的 Kubernetes client 包。在 config.NewSourceAPI 的方法里，启动了一个协程周期性地调用 watcher 的 list 与 Watch 方法获取数据，然后转换成 ServiceUpdate 事件，写入 Channel 中。下面是关键源码：

```
func (s *servicesReflector) run(resourceVersion *string) {
    if len(*resourceVersion) == 0 {
        services, err := s.watcher.List(labels.Everything())
        if err != nil {
            glog.Errorf("Unable to load services: %v", err)
            // TODO: reconcile with pkg/client/cache which doesn't use reflector.
            time.Sleep(wait.Jitter(s.waitDuration, 0.0))
            return
        }
        *resourceVersion = services.ResourceVersion
        // TODO: replace with code to update the
        s.services <- ServiceUpdate{Op: SET, Services: services.Items}
    }
}
```

```

watcher, err := s.watcher.Watch(labels.Everything(), fields.Everything(),
*resourceVersion)
if err != nil {
    glog.Errorf("Unable to watch for services changes: %v", err)
    if !client.IsTimeout(err) {
        // Reset so that we do a fresh get request
        *resourceVersion = ""
    }
    time.Sleep(wait.Jitter(s.waitDuration, 0.0))
    return
}
defer watcher.Stop()
ch := watcher.ResultChan()
s.watchHandler(resourceVersion, ch, s.services)
}

```

在上面的代码中，初始时资源版本变量 resourceVersion 为空，于是会执行 Service 的全量拉取动作（watcher.List），之后 Watch 资源会开始发生变化（watcher.Watch）并将 Watch 的结果（一个 Channel 保持了 Service 的变动数据）也转换为对应的 ServiceUpdate 事件并写入 Channel 中。另外，当拉取数据的调用发生异常时，resourceVersion 恢复为空，导致重新进行全量资源的拉取动作。这种自修复能力的编程设计足以见证谷歌大神们的深厚编程功力；另外，笔者认为 kube-proxy 这里的 ServiceConfig 的设计实现思路和代码要比 kubelet 中的好一点，虽然两个作者都是顶尖高手。

接下来才开始进入本节的重点，即服务代理的实现机制分析。首先，我们从代码中的 load balance 组件说起。下面是 kube-proxy 中定义的 Load Balancer 接口：

```

type LoadBalancer interface {
    NextEndpoint(service ServicePortName, srcAddr net.Addr) (string, error)
    NewService(service ServicePortName, sessionAffinityType api.ServiceAffinity,
stickyMaxAgeMinutes int) error
    CleanupStaleStickySessions(service ServicePortName)
}

```

LoadBalancer 有 3 个接口，其中 NextEndpoint 方法用于给访问指定 Service 的新客户端请求分配一个可用的 Endpoint 地址；NewService 用来添加一个新服务到负载均衡器上；CleanupStaleStickySessions 则用来清理过期的 Session 会话。目前 kube-proxy 只实现了一个基于 round-robin 算法的负载均衡器，它就是 proxy.LoadBalancerRR 组件。

LoadBalancerRR 采用了 affinityState 这个结构体来保存当前客户端的会话信息，然后在 affinityPolicy 里用一个 Map 来记录（属于某个 Service 的）所有活动的客户端会话，这是它实现 Session 亲和性的负载均衡调度的基础。

```

type affinityState struct {
    clientIP string
}

```

```

    //clientProtocol api.Protocol //not yet used
    //sessionCookie string      //not yet used
    endpoint string
    lastUsed time.Time
}
type affinityPolicy struct {
    affinityType api.ServiceAffinity
    affinityMap map[string]*affinityState // map client IP -> affinity info
    ttlMinutes int
}

```

`balancerState` 用来记录一个 Service 的所有 Endpoint(数组)、当前所使用的 Endpoint 的 index，以及对应的所有活动的客户端会话 (affinityPolicy)。其定义如下：

```

type balancerState struct {
    endpoints []string // a list of "ip:port" style strings
    index     int      // current index into endpoints
    affinity   affinityPolicy
}

```

有了上面的认识，再看 `LoadBalancerRR` 的构造函数就简单多了，它内部用一个 map 记录每个服务的 `balancerState` 状态，当然初始化时还是空的：

```

func NewLoadBalancerRR() *LoadBalancerRR {
    return &LoadBalancerRR{
        services: map[ServicePortName]*balancerState{},
    }
}

```

`LoadBalancerRR` 的 `NewService` 方法代码很简单，就是在它的 `services` 里增加一个记录项，用户端的会话超时时间 `ttlMinutes` 默认为 3 小时，下面是相关源码：

```

func (lb *LoadBalancerRR) NewService(svcPort ServicePortName, affinityType
api.ServiceAffinity, ttlMinutes int) error {
    lb.lock.Lock()
    defer lb.lock.Unlock()
    lb.newServiceInternal(svcPort, affinityType, ttlMinutes)
    return nil
}
func (lb *LoadBalancerRR) newServiceInternal(svcPort ServicePortName, affinityType
api.ServiceAffinity, ttlMinutes int) *balancerState {
    if ttlMinutes == 0 {
        ttlMinutes = 180
    }
    if _, exists := lb.services[svcPort]; !exists {
        lb.services[svcPort] = &balancerState{affinity:
*newAffinityPolicy(affinityType, ttlMinutes)}
        glog.V(4).Infof("LoadBalancerRR service %q did not exist, created",
        svcPort)
    }
}

```

```

    } else if affinityType != "" {
        lb.services[svcPort].affinity.affinityType = affinityType
    }
    return lb.services[svcPort]
}

```

我们在前面提到过 ServiceConfig 同步并监听 API Server 上的 api.Service 的数据变化，然后调用 Listener（proxy.Proxier 是 ServiceConfig 唯一注册的 Listener）的 OnUpdate 接口完成通知。而上述 NewService 就是在 proxy.Proxier 的 OnUpdate 方法里被调用的，从而实现了 Service 自动添加到 LoadBalancer 的机制。

我们再来看 LoadBalancerRR 的 NextEndpoint 方法，它实现了经典的 round-robin 负载均衡算法。NextEndpoint 方法首先判断当前服务是否有保持会话（sessionAffinity）的要求，如果有，则看当前请求是否有连接可用：

```

if sessionAffinityEnabled {
    // Caution: don't shadow ipaddr
    var err error
    ipaddr, _, err = net.SplitHostPort(srcAddr.String())
    if err != nil {
        return "", fmt.Errorf("malformed source address %q: %v", srcAddr.
String(), err)
    }
    sessionAffinity, exists := state.affinity.affinityMap[ipaddr]
    if exists && int(time.Now().Sub(sessionAffinity.lastUsed).Minutes()) <
state.affinity.ttlMinutes {
        // Affinity wins.
        endpoint := sessionAffinity.endpoint
        sessionAffinity.lastUsed = time.Now()
        glog.V(4).Infof("NextEndpoint for service %q from IP %s with
sessionAffinity %+v: %s", svcPort, ipaddr, sessionAffinity, endpoint)
        return endpoint, nil
    }
}

```

如果服务无须会话保持、新建会话及会话过期，则采用 round-robin 算法得到下一个可用的服务端口，如果服务有会话保持需求，则保存当前的会话状态：

```

// Take the next endpoint.
endpoint := state.endpoints[state.index]
state.index = (state.index + 1) % len(state.endpoints)
if sessionAffinityEnabled {
    var affinity *affinityState
    affinity = state.affinity.affinityMap[ipaddr]
    if affinity == nil {
        affinity = new(affinityState) //&affinityState{ipaddr, "TCP", "",
endpoint, time.Now()}
    }
}

```

```

        state.affinity.affinityMap[ipaddr] = affinity
    }
    affinity.lastUsed = time.Now()
    affinity.endpoint = endpoint
    affinity.clientIP = ipaddr
    glog.V(4).Infof("Updated affinity key %s: %+v", ipaddr, state.affinity.
affinityMap[ipaddr])
}
return endpoint, nil

```

接下来我们看看 Service 的 Endpoint 信息是如何添加到 LoadBalancerRR 上的？答案很简单，类似之前我们分析过的 ServiceConfig。kube-proxy 也设计了一个 EndpointsConfig 来拉取和监听 API Server 上的服务的 Endpoint 信息，并调用 LoadBalancerRR 的 OnUpdate 接口完成通知，在这个方法里，LoadBalancerRR 完成了服务访问端口的添加和同步逻辑。

我们先来看看 api.Endpoints 的定义：

```

type EndpointAddress struct {
    IP string
    TargetRef *ObjectReference
}
type EndpointPort struct {
    Name string
    Port int
    Protocol Protocol
}
type EndpointSubset struct {
    Addresses []EndpointAddress
    Ports     []EndpointPort
}
type Endpoints struct {
    TypeMeta   `json: ",inline"`
    ObjectMeta `json: "metadata,omitempty"`
    Subsets   []EndpointSubset
}

```

一个 EndpointAddress 与 EndpointPort 对象可以组成一个服务访问地址，而在 EndpointSubset 对象里则定义了两个单独的 EndpointAddress 与 EndpointPort 数组而不是“服务访问地址”的一个列表。初看这样的定义你可能会觉得很奇怪，为什么没有设计一个 Endpoint 结构？这里的深层次原因在于，Service 的 Endpoint 信息来源于两个独立的实体：Pod 与 Service，前者负责提供 IP 地址即 EndpointAddress，而后者负责提供 Port 即 EndpointPort。由于在一个 Pod 上可以运行多个 Service，而一个 Service 也通常跨越多个 Pod，于是就产生了一个“笛卡尔乘积”的 Endpoint 列表，这就是 EndpointSubset 的设计灵感。

举例说明，对于如下表示的 EndpointSubset：

```
{
    Addresses: [{"ip": "10.10.1.1"}, {"ip": "10.10.2.2"}],
    Ports: [{"name": "a", "port": 8675}, {"name": "b", "port": 309}]
}
```

会产生如下 Endpoint 列表：

```
a: [ 10.10.1.1:8675, 10.10.2.2:8675 ],
b: [ 10.10.1.1:309, 10.10.2.2:309 ]
```

LoadBalancerRR 的 OnUpdate 方法里循环对每个 api.Endpoints 进行处理，先把它转化为一个 Map，Map 的 Key 是 EndpointPort 的 Name 属性（代表一个 Service 的访问端口）；而 Value 则是 hostPortPair 的一个数组，hostPortPair 其实就是之前缺失的 Endpoint 结构体，包括一个 IP 地址与端口属性，即某个服务在一个 Pod 上的对应访问端口。

```
portsToEndpoints := map[string][]hostPortPair{}
for i := range svcEndpoints.Subsets {
    ss := &svcEndpoints.Subsets[i]
    for i := range ss.Ports {
        port := &ss.Ports[i]
        for i := range ss.Addresses {
            addr := &ss.Addresses[i]
            portsToEndpoints[port.Name] = append(portsToEndpoints
[port.Name], hostPortPair{addr.IP, port.Port})
                // Ignore the protocol field - we'll get that from the Service
objects.
        }
    }
}
```

下一步，针对 portsToEndpoints 进行循环处理。对于每个记录，判断是否已经在 services 中存在，并做出相应的更新或跳过的逻辑，最后删除那些已经不在集合中的端口，完成整个同步逻辑。下面是相关代码：

```
for portname := range portsToEndpoints {
    svcPort := ServicePortName{types.NamespacedName{svcEndpoints.Namespace,
svcEndpoints.Name}, portname}
    state, exists := lb.services[svcPort]
    curEndpoints := []string{}
    if state != nil {
        curEndpoints = state.endpoints
    }
    newEndpoints := flattenValidEndpoints(portsToEndpoints[portname])

    if !exists || state == nil || len(curEndpoints) != len(newEndpoints)
|| !slicesEquiv(slice.CopyStrings(curEndpoints), newEndpoints) {
        glog.V(1).Infof("LoadBalancerRR: Setting endpoints for %s to %v",
svcPort, newEndpoints)
```

```

        lb.updateAffinityMap(svcPort, newEndpoints)
        // OnUpdate can be called without NewService being called externally.
        // To be safe we will call it here. A new service will only be created
        // if one does not already exist. The affinity will be updated
        // later, once NewService is called.
        state = lb.newServiceInternal(svcPort, api.ServiceAffinity(""), 0)
        state.endpoints = slice.ShuffleStrings(newEndpoints)

        // Reset the round-robin index.
        state.index = 0
    }
    registeredEndpoints[svcPort] = true
}
}

// Remove endpoints missing from the update.
for k := range lb.services {
    if _, exists := registeredEndpoints[k]; !exists {
        glog.V(2).Infof("LoadBalancerRR: Removing endpoints for %s", k)
        delete(lb.services, k)
    }
}
}

```

LoadBalancerRR 的代码总体来说还是比较简单的, 它主要被 kube-proxy 中的关键组件 proxy. Proxier 所使用, 后者用到的主要数据结构为 proxy.serviceInfo, 它定义和保存了一个 Service 的代理过程中的必要参数和对象。下面是其定义:

```

type serviceInfo struct {
    portal          portal
    protocol       api.Protocol
    proxyPort      int
    socket         proxySocket
    timeout        time.Duration
    nodePort       int
    loadBalancerStatus api.LoadBalancerStatus
    sessionAffinityType api.ServiceAffinity
    stickyMaxAgeMinutes int
    // Deprecated, but required for back-compat (including e2e)
    deprecatedPublicIPs []string
}

```

serviceInfo 的各个属性解释如下。

- ◎ **portal:** 用于存放服务的 Portal 地址, 即 Service 的 Cluster IP (VIP) 地址与端口。
- ◎ **protcal:** 服务的 TCP, 目前是 TCP 与 UDP。
- ◎ **socket、proxyPort:** socket 是 Proxier 在本机上为该服务打开的代理 Socket; proxyPort 则是这个代理 Socket 的监听端口。

- ◎ `timeout`: 目前只用于 UDP 的 Service, 表明服务“链接”的超时时间。
- ◎ `nodePort`: 该服务定义的 NodePort。
- ◎ `loadBalancerStatus`: 在 Cloud 环境下, 如果存在由 Cloud 服务提供者提供的负载均衡器(软件或硬件)用作 Kubernetes Service 的负载均衡, 则这里存放这些负载均衡器的 IP 地址。
- ◎ `sessionAffinityType`: 该服务的负载均衡调度是否保持会话。
- ◎ `stickyMaxAgeMinutes`: 即前面说的 Session 过期时间。
- ◎ `deprecatedPublicIPs`: 已过期、废弃的服务的 Public IP 地址。

理解了 `serviceInfo`, 我们再来看 Proxier 的数据结构:

```
type Proxier struct {
    loadBalancer LoadBalancer
    mu           sync.Mutex // protects serviceMap
    serviceMap   map[ServicePortName]*serviceInfo
    portMapMutex sync.Mutex
    portMap     map[portMapKey]ServicePortName
    numProxyLoops int32
    listenIP     net.IP
    iptables     iptables.Interface
    hostIP       net.IP
    proxyPorts   PortAllocator
}
```

Proxier 用一个 Map 维护了每个服务的 `serviceInfo` 信息, 同时为了快速查询和检测服务端口是否有冲突, 比如定义了两个一样端口的服务, 又设计了一个 `portMap`, 其 Key 为服务的端口信息(`portMapKey` 由 `port` 和 `protocol` 组合而成), value 为 `ServicePortName`。Proxier 的 `listenIP` 为 Proxier 监听的本节点 IP, 它在这个 IP 上接收请求并做转发代理。由于每个服务的 `proxySocket` 在本节点监听的 Port 端口默认是系统随机分配的, 所以使用 `PortAllocator` 来分配这个端口。另外, Service 的 Portal 与 NodePort 是通过 Linux 防火墙机制来实现的, 因此这里引用了 Iptables 的组件完成相关操作。

要想理解 Proxier 中使用 Iptables 的方式, 首先我们要弄明白 Kubernetes 中 Service 访问的一些网络细节。先来看看图 6.9, 这是一个外部应用通过 NodePort (`TCP://NodeIP:NodePort`) 来访问 Service 时的网络流量示意图。访问流量进入节点网卡 `eth0` 后, 到达 Iptables 的 `PREROUTING` 链, 通过 `KUBE-NODEPORT-CONTAINER` 这个 NAT 规则被转发到 `kube-proxy` 进程上该 Service 对应的 Proxy 端口, 然后由 `kube-proxy` 进程进行负载均衡并且将流量转发到 Service 所在 Container 的本地端口。

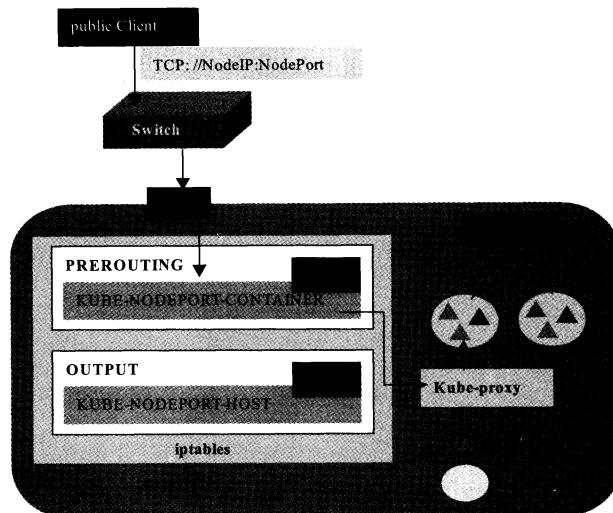


图 6.9 外部应用通过 NodePort 访问 Service 的网络流量示意图

根据 Iptables 的机制，本地进程发起的流量会经过 Iptables 的 OUTPUT 链，于是 kube-proxy 在这里也增加了相同作用的 NAT 规则：KUBE-NODEPORT-HOST。这样一来，如果本地容器内的进程以 NodePort 方式来访问 Service，则流量也会被转发到 kube-proxy 上，虽然以这种方式访问的情况比较少见。

服务之间通过 Service Portal 方式访问的流量转发机制跟 NodePort 方式在本质上是一样的，也是通过 NAT，如图 6.10 所示。当 Service A 用 Service B 的 Portal 地址去访问时，流量经过 Iptables 的 OUTPUT 链经 NAT 规则 KUBE-PORTALS-HOST 的转换被转发到 kube-proxy 上，然后被转发给 Service B 所在的容器。

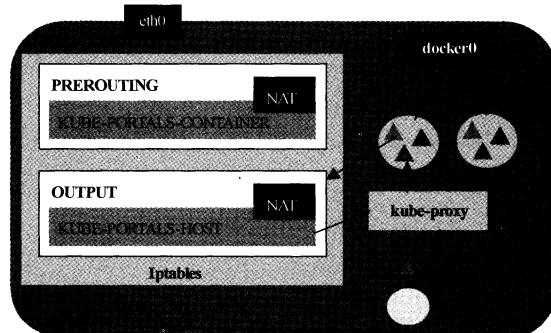


图 6.10 以 Service Portal 方式访问 Service 的流量示意图

Proxier 在创建 Iptables 的 PREROUTING 链中的 NAT 转发规则时，有一些特殊性，源码作者在代码中做了如下注释：

“这是一个复杂的问题。

如果 Proxy 的 Proxier.listenIP 设置为 0.0.0.0，即绑定到所有端口上，那么我们采用 REDIRECT 这种方式进行流量转发，因为这种情况下，返回的流量与进入的流量使用同一个网络端口，这就满足了 NAT 的规则。其他情况则采用 DNAT 转发流量，但 DNAT 到 127.0.0.1 时，流量会消失，这似乎是 Iptables 的一个众所周知的问题，所以这里不允许 Proxy 绑定到 localhost 上。”

现在再看下面这段代码就容易理解了，用来生成 KUBE-NODEPORT-CONTAINER 这条 NAT 规则：

```
func (proxier *Proxier) iptablesContainerNodePortArgs(nodePort int, protocol
api.Protocol, proxyIP net.IP, proxyPort int, service ServicePortName) []string {
    args := iptablesCommonPortalArgs(nil, nodePort, protocol, service)
    if proxyIP.Equal(zeroIPv4) || proxyIP.Equal(zeroIPv6) {
        // TODO: Can we REDIRECT with IPv6?
        args = append(args, "-j", "REDIRECT", "--to-ports", fmt.Sprintf("%d",
proxyPort))
    } else {
        // TODO: Can we DNAT with IPv6?
        args = append(args, "-j", "DNAT", "--to-destination", net.JoinHostPort
(proxyIP.String(), strconv.Itoa(proxyPort)))
    }
    return args
}
```

弄明白 Proxier 中关于 Iptables 的事情之后，我们来研究分析下 Proxier 如何在 OnUpdate 方法里为每个 Service 建立起对应的 Proxy 并完成同步工作。首先，在 OnUpdate 方法里创建一个 map (activeServices) 来标识当前所有 alive 的 Service，key 为 ServicePortName，然后对 OnUpdate 参数里的 Service 数组进行循环，判断每个 Service 是否需要进行新建、变更或者删除操作，对于需要新建或者变更的 Service，先用 PortAllocator 获取一个新的未用的本地代理端口，然后调用 addServiceOnPort 方法创建一个 ProxySocket 用于实现此服务的代理，接着调用 openPortal 方法添加 iptables 里的 NAT 映射规则，最后调用 LoadBalancer 的 NewService 方法把该服务添加到负载均衡器上。OnUpdate 方法的最后一段逻辑是处理已经被删除的 Service，对于每个要被删除的 Service，先删除 Iptables 中相关的 NAT 规则，然后关闭对应的 proxySocket，最后释放 ProxySocket 占用的监听端口并将该端口“还给” PortAllocator。

从上面的分析中，我们看到 addServiceOnPort 是 Proxier 的核心方法之一。下面是该方法的源码：

```
func (proxier *Proxier) addServiceOnPort(service ServicePortName, protocol
api.Protocol, proxyPort int, timeout time.Duration) (*serviceInfo, error) {
    sock, err := newProxySocket(protocol, proxier.listenIP, proxyPort)
    if err != nil {
        return nil, err
    }
```

```

    , portStr, err := net.SplitHostPort(sock.Addr().String())
    if err != nil {
        sock.Close()
        return nil, err
    }
    portNum, err := strconv.Atoi(portStr)
    if err != nil {
        sock.Close()
        return nil, err
    }
    si := &serviceInfo{
        proxyPort:          portNum,
        protocol:           protocol,
        socket:             sock,
        timeout:            timeout,
        sessionAffinityType: api.ServiceAffinityNone, // default
        stickyMaxAgeMinutes: 180,                      // TODO: paramaterize this
in the API.
    }
    proxier.setServiceInfo(service, si)

    glog.V(2).Infof("Proxying for service %q on %s port %d", service, protocol,
portNum)
    go func(service ServicePortName, proxier *Proxier) {
        defer util.HandleCrash()
        atomic.AddInt32(&proxier.numProxyLoops, 1)
        sock.ProxyLoop(service, si, proxier)
        atomic.AddInt32(&proxier.numProxyLoops, -1)
    }(service, proxier)

    return si, nil
}

```

在上述代码中，先创建一个 `ProxySocket`，然后创建一个 `serviceInfo` 并添加到 `Proxier` 的 `serviceMap` 中，最后启动一个协程调用 `ProxySocket` 的 `ProxyLoop` 方法，使得 `ProxySocket` 进入 `Listen` 状态，开始接收并转发客户端请求。

`kube-proxy` 中的 `ProxySocket` 有两个实现，其中一个是 `tcpProxySocket`，另外一个是 `udpProxySocket`，二者的工作原理都一样，它们的工作流程就是为每个客户端 `Socket` 请求创建一个到 `Service` 的后端 `Socket` 连接，并且“打通”这两个 `Socket`，即把客户端 `Socket` 发来的数据“复制”到对应的后端 `Socket` 上，然后把后端 `Socket` 上服务响应的数据写入客户端 `Socket` 上去。

以 `tcpProxySocket` 为例，我们先看看它是如何完成 `Service` 后端连接创建过程的：

```

func tryConnect(service ServicePortName, srcAddr net.Addr, protocol string,
proxier *Proxier) (out net.Conn, err error) {

```

```

        for _, retryTimeout := range endpointDialTimeout {
            endpoint, err := proxier.loadBalancer.NextEndpoint(service, srcAddr)
            if err != nil {
                glog.Errorf("Couldn't find an endpoint for %s: %v", service, err)
                return nil, err
            }
            glog.V(3).Infof("Mapped service %q to endpoint %s", service, endpoint)
            outConn, err := net.DialTimeout(protocol, endpoint, retryTimeout*time.Second)
            if err != nil {
                if isTooManyFDsError(err) {
                    panic("Dial failed: " + err.Error())
                }
                glog.Errorf("Dial failed: %v", err)
                continue
            }
            return outConn, nil
        }
        return nil, fmt.Errorf("failed to connect to an endpoint. ")
    }
}

```

在上述方法里，首先调用 `loadBalancer.NextEndpoint` 方法获取服务的下一个可用 Endpoint 地址，然后调用标准网络库中的方法建立到此地址的连接，如果连接失败，则会重新尝试，间隔时间指数增加（参见 `endpointDialTimeout` 的值）。

在后端 Service 的连接建立以后，`proxyTCP` 方法就会启动两个协程，通过调用 Go 标准库 `io` 里的 `Copy` 方法把输入流的数据写入输出流，从而完成前后端连接的数据转发功能。此外，`proxyTCP` 方法会阻塞，直到前后端两个连接的数据流都关闭（或结束）才会返回。下面是其源码：

```

func proxyTCP(in, out *net.TCPConn) {
    var wg sync.WaitGroup
    wg.Add(2)
    glog.V(4).Infof("Creating proxy between %v <-> %v <-> %v <-> %v",
        in.RemoteAddr(), in.LocalAddr(), out.LocalAddr(), out.RemoteAddr())
    go copyBytes("from backend", in, out, &wg)
    go copyBytes("to backend", out, in, &wg)
    wg.Wait()
    in.Close()
    out.Close()
}

```

这里我们留一个问题，`kube-proxy` 会在当前节点上为每个 Service 都建立一个代理么？不管本节点上是否有该 Service 对应的 Pod？

6.6.3 设计总结

从之前的启动流程和代码分析来看，kube-proxy 的设计和实现还是比较精巧和紧凑的，它的流程只有一个：从 Kubernetes API Server 上同步 Service 及其 Endpoint 信息，为每个 Service 建立一个本地代理以完成具备负载均衡能力的服务转发功能。图 6.11 给出了 kube-proxy 的总体设计示意图，为了清晰地表明整个业务流程和数据传递方向，这里省去了一些非关键的结构体和对象。`app.ProxyServer` 创建了一个 `config.SourceAPI` 的结构体，用于拉取 Kubernetes API Server 上的 Service 与 Endpoint 配置信息，分别由 `config.servicesReflector` 与 `config.endpointsReflector` 这两个对象来实现，它们各自通过相应的 Kubernetes Client API 来拉取数据并且生成对应的 Update 信息放入 Channel 中，最终 Channel 中的 Service 数据到达 `proxy.Proxier` 上，`proxy.Proxier` 为每个 Service 建立一个 `proxySocket` 实现服务代理并且在 iptables 上创建相关的 NAT 规则，然后在 LoadBalancer 组件上开通该服务的负载均衡功能；而 Channel 中的 Endpoints 数据则被发送

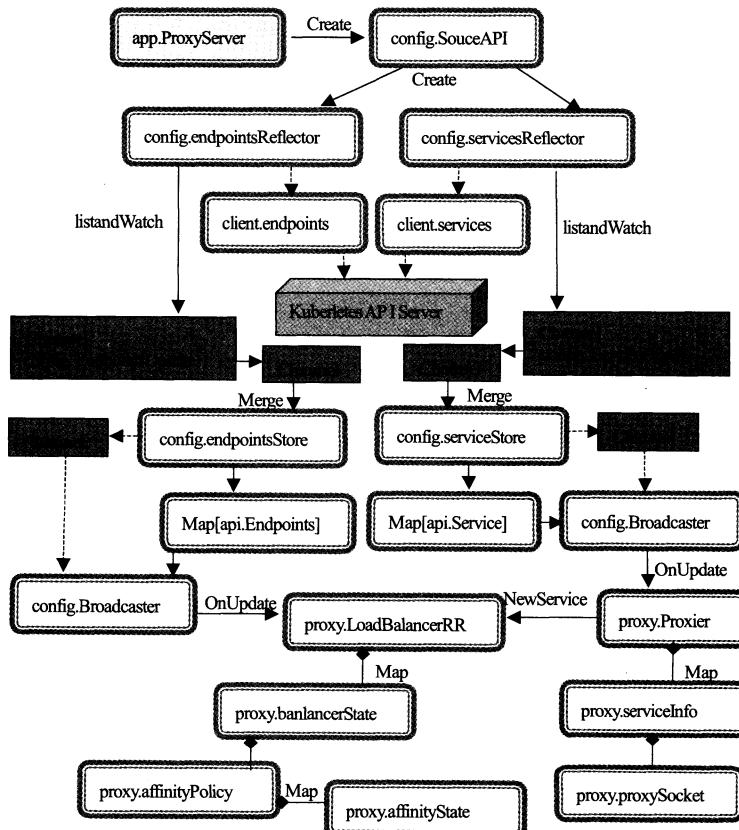


图 6.11 与 kubelet 总体相关的设计示意图

到 `proxy.LoadBalancerRR` 组件，用于给每个服务建立一个负载均衡的状态机，每个服务用 `banlancerState` 结构体来保存该服务可用的 `Endpoint` 地址及当前的会话状态 `affinityPolicy`，对于需要保存会话状态的服务，`affinityPolicy` 用一个 `Map` 来存储每个客户的会话状态 `affinityState`。

6.7 kubectl 进程源码分析

`kubectl` 与之前的 `Kubernetes` 进程不同，它不是一个后台运行的守护进程，而是 `Kubernetes` 提供的一个命令行工具（CLI），它提供了一组命令来操作 `Kubernetes` 集群。

`kubectl` 进程的入口类源码位置如下：

```
github.com/GoogleCloudPlatform/kubernetes/cmd/kubectl/kubectl.go
```

入口 `main()` 函数的逻辑很简单：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    cmd := cmd.NewKubectlCommand(cmdutil.NewFactory(nil), os.Stdin, os.Stdout,
        os.Stderr)
    if err := cmd.Execute(); err != nil {
        os.Exit(1)
    }
}
```

上述代码通过 `NewKubectlCommand` 方法创建了一个具体的 `Command` 命令并调用它的 `Execute` 方法执行，这是工厂模式结合命令模式的一个经典设计案例。从 `NewKubectlCommand` 的源码中可以看到，`kubectl` 的 CLI 命令框架使用了 GitHub 开源项目（<https://github.com/spf13/cobra>），下面是该框架中对 `Command` 的定义：

```
type Command struct {
    Use string // The one-line usage message.
    Short string // The short description shown in the 'help' output.
    Long string // The long message shown in the 'help <this-command>' output.
    Run func(cmd *Command, args []string) // Run runs the command.
}
```

实现一个具体 `Command` 就只要实现 `Command` 的 `Run` 函数即可，下面是其官方网页给出的一个 `Echo` 命令的例子：

```
var cmdEcho = &cobra.Command{
    Use:   "echo [string to echo] ",
    Short: "Echo anything to the screen",
    Long:  `echo is for echoing anything back.
    Echo works a lot like print, except it has a child command.
`,
```

```

Run: func(cmd *cobra.Command, args []string) {
    fmt.Println("Print: " + strings.Join(args, " "))
},
}
}

```

由于大多数 kubectl 的命令都需要访问 Kubernetes API Server，所以 kubectl 设计了一个类似命令的上下文环境的对象——util.Factory 供 Command 对象使用。

在接下来的几个章节中，我们对 kubectl 中的几个典型 Command 的源码逐一解读。

6.7.1 kubectl create 命令

kubectl create 命令通过调用 Kubernetes API Server 提供的 Rest API 来创建 Kubernetes 资源对象，例如 Pod、Service、RC 等，资源的描述信息来自-f 指定的文件或者来自命令行的输入流。下面是创建 create 命令的相关源码：

```

func NewCmdCreate(f *cmdutil.Factory, out io.Writer) *cobra.Command {
    var filenames util.StringList
    cmd := &cobra.Command{
        Use:      "create -f FILENAME",
        Short:    "Create a resource by filename or stdin",
        Long:     "create long",
        Example:  "create_example",
        Run: func(cmd *cobra.Command, args []string) {
            cmdutil.CheckErr(ValidateArgs(cmd, args))
            cmdutil.CheckErr(RunCreate(f, out, filenames))
        },
    }
    usage := "Filename, directory, or URL to file to use to create the resource"
    kubectl.AddJsonFilenameFlag(cmd, &filenames, usage)
    cmd.MarkFlagRequired("filename")
    return cmd
}

```

AddJsonFilenameFlag 方法限制 filename 参数 (-f) 的文件名后缀只能是 json、yaml 或者 yml 中的一种，并且将参数值填充到 filenames 这个 Set 集合中，随后被 Command 的 Run 函数中的 RunCreate 方法所引用，后者就是 kubectl create 命令的核心逻辑所在。

RunCreate 方法使用到了 resource.Builder 对象，它是 kubectl 中的一处复杂设计，采用了 Visitor 的设计模式，kubectl 的很多命令都用到了它。Builder 的目标是根据命令行输入的资源相关的参数，创建针对性的 Visitor 对象来获取对应的资源，最后遍历相关的所有 Visitor 对象，触发用户指定的 VisitorFun 回调函数来处理每个具体的资源，最终完成资源对象的业务处理逻辑。由于涉及的资源参数有各种情况，所以导致 Builder 的代码很复杂。以下是 Builder 所能操作的

各种资源参数：

- ◎ 通过输入流提供具体的资源描述；
- ◎ 通过本地文件内容或者 HTTP URL 的输出流来获取资源描述；
- ◎ 文件列表提供多个资源描述；
- ◎ 指定资源类型，通过查询 Kubernetes API Server 来获取相关类型的资源；
- ◎ 指定资源的 selector 条件如 cluster-service=true，查询 Kubernetes API Server 来获取相关的资源；
- ◎ 指定资源的 namespace 来查询符合条件的相关资源。

下面是 `resource.Builder` 的定义：

```
type Builder struct {
    mapper *Mapper
    errs []error
    paths []Visitor
    stream bool
    dir    bool
    selector labels.Selector
    selectAll bool
    resources []string
    namespace string
    names   []string
    resourceTuples []resourceTuple
    defaultNamespace bool
    requireNamespace bool
    flatten bool
    latest  bool
    requireObject bool
    singleResourceType bool
    continueOnError  bool
    schema validation.Schema
}
```

其实 `Builder` 很像一个 SQL 查询条件的生成器，里面包括了各种“查询”条件，在指定不同的查询条件时，会生成不同的 `Visitor` 接口来处理这些查询条件，最后遍历所有 `Visitor`，就得到最终的“查询结果”。`Builder` 返回的 `Result` 对象里也包括 `Visitor` 对象及可能的最终资源列表等信息，由于资源查询存在各种情况，所以 `Result` 也提供了多种方法，比如还包括了 `Watch` 资源变化的方法。

`RunCreate` 方法里先创建了一个 `Builder`，设置各种必要参数，然后调用 `Builder` 的 `Do` 方法，返回一个 `Result`，代码如下：

```

schema, err := f.Validator()
mapper, typer := f.Object()
r := resource.NewBuilder(mapper, typer, f.ClientMapperForCommand()).
    Schema(schema).
    ContinueOnError().
    NamespaceParam(cmdNamespace).DefaultNamespace().
    FilenameParam(enforceNamespace, filenames...).
    Flatten().
    Do()

```

其中，`schema` 对象用来校验资源描述是否正确，比如有没有缺少字段或者属性的类型错误等；`mapper` 对象用来完成从资源描述信息到资源对象的转换，用来在 REST 调用过程中完成数据转换；`FilenameParam` 是这里唯一指定 `Builder` 的资源参数的方法，即把命令行传入的 `filenames` 参数作为资源参数；`Flatten` 方法则告诉 `Builder`，这里的资源对象其实是一个数组，需要 `Builder` 构造一个 `FlattenListVisitor` 来遍历 `Visit` 数组中的每个资源项目；`Do` 方法则返回一个 `Rest` 对象，里面包括与资源相关的 `Visitor` 对象。

下面是 `NamespaceParam` 方法的源码，主要逻辑为调用 `Builder` 的 `Builder.Stdin`、`Builder.URL` 或 `Builder.Path` 方法来处理不同类型的资源参数，这些方法会生成对应的 `Visitor` 对象并加入 `Builder` 的 `Visitor` 数组里（`paths` 属性）。

```

func (b *Builder) FilenameParam(enforceNamespace bool, paths ...string) *Builder {
    for _, s := range paths {
        switch {
        case s == "-":
            b.Stdin()
        case strings.Index(s, "http:// ") == 0 || strings.Index(s, "https:// ") == 0:
            url, err := url.Parse(s)
            if err != nil {
                b.errs = append(b.errs, fmt.Errorf("the URL passed to filename %q is not valid:
%v", s, err))
                continue
            }
            b.URL(url)
        default:
            b.Path(s)
        }
    }
    if enforceNamespace {
        b.RequireNamespace()
    }
    return b
}

```

不管是标准输入流、URL，还是文件目录或者文件本身，这里处理资源的 `Visitor` 都是

StreamVisitor 这个实现（FileVisitor 与 FileVisitorForSTDIN 是 StreamVisitor 的一个 Wrapper）。下面是 StreamVisitor 的 Visit 接口代码：

```
func (v *StreamVisitor) Visit(fn VisitorFunc) error {
    d := yaml.NewYAMLorJSONDecoder(v.Reader, 4096)
    for {
        ext := runtime.RawExtension{}
        if err := d.Decode(&ext); err != nil {
            if err == io.EOF {
                return nil
            }
            return err
        }
        ext.RawJSON = bytes.TrimSpace(ext.RawJSON)
        if len(ext.RawJSON) == 0 || bytes.Equal(ext.RawJSON, []byte("null")) {
            continue
        }
        if err := ValidateSchema(ext.RawJSON, v.Schema); err != nil {
            return err
        }
        info, err := v.InfoForData(ext.RawJSON, v.Source)
        if err != nil {
            if v.IgnoreErrors {
                fmt.Fprintf(os.Stderr, "error: could not read an encoded object from
%s: %v\n", v.Source, err)
                glog.V(4).Infof("Unreadable: %s", string(ext.RawJSON))
                continue
            }
            return err
        }
        if err := fn(info); err != nil {
            return err
        }
    }
}
```

在上述代码中，首先从输入流中解析具体的资源对象，然后创建一个 Info 结构体进行包装（转换后的资源对象存储在 Info 的 Object 属性中），最后再用这个 Info 对象作为参数调用回调函数 VisitorFunc，从而完成整个逻辑流程。下面是 RunCreate 方法里调用 Builder 的 Visit 方法触发 Visitor 执行时的源码，可以看到这里的 VisitorFunc 所做的事情是通过 Rest Client 发起 Kubernetes API 调用，把资源对象写入资源注册表里：

```
err = r.Visit(func(info *resource.Info) error {
    data, err := info.Mapping.Codec.Encode(info.Object)
    if err != nil {
        return cmdutil.AddSourceToErr("creating", info.Source, err)
    }
    if err := r.Client.Create(context.TODO(), data, &meta_v1.Status{}); err != nil {
        return cmdutil.AddSourceToErr("creating", info.Source, err)
    }
})
```

```

        }
        obj, err := resource.NewHelper(info.Client, info.Mapping).Create(info.
Namespace, true, data)
        if err != nil {
            return cmdutil.AddSourceToErr("creating", info.Source, err)
        }
        count++
        info.Refresh(obj, true)
        printObjectSpecificMessage(info.Object, out)
        fmt.Fprintf(out, "%s/%s\n", info.Mapping.Resource, info.Name)
        return nil
    })
}

```

6.7.2 rolling-update 命令

kubectl rolling-update 命令负责滚动更新（升级）RC（ReplicationController），下面是创建对应 Command 的源码：

```

func NewCmdRollingUpdate(f *cmdutil.Factory, out io.Writer) *cobra.Command {
    cmd := &cobra.Command{
        Use:   "rolling-update OLD_CONTROLLER_NAME ([NEW_CONTROLLER_NAME] -image
=NEW_CONTAINER_IMAGE | -f NEW_CONTROLLER_SPEC) ",
        // rollingupdate is deprecated.
        Aliases: []string{"rollingupdate"},
        Short:   "Perform a rolling update of the given ReplicationController. ",
        Long:    "rollingUpdate_long",
        Example: "rollingUpdate_example",
        Run: func(cmd *cobra.Command, args []string) {
            err := RunRollingUpdate(f, out, cmd, args)
            cmdutil.CheckErr(err)
        },
    }
    cmd.Flags().String("update-period", updatePeriod, `Time to wait between
updating pods. Valid time units are "ns", "us" (or "μs"), "ms", "s", "m", "h".`)
}

```

此处省去一些命令参数添加的非关键代码：

```

    cmdutil.AddPrinterFlags(cmd)
    return cmd
}

```

从上述代码中我们看到 rolling-update 命令的执行函数为 RunRollingUpdate，在分析这个函数之前，我们先了解下 rolling-update 执行过程中的一个关键逻辑。

rolling update 动作可能由于网络超时或者用户等得不耐烦等原因被中断，因此我们可能会重复执行一条 rolling-update 命令，目的只有一个，就是恢复之前的 rolling update 动作。为了实

现这个目的，`rolling-update` 程序在执行过程中会在当前 `rolling-update` 的 RC 上增加一个 `Annotation` 标签——`kubectl.kubernetes.io/next-controller-id`，标签的值就是下一个要执行的新 RC 的名字。此外，对于 Image 升级这种更新方式，还会在 RC 的 Selector 上 (`RC.Spec.Selector`) 贴一个名为 `deploymentKey` 的 Label，Label 的值是 RC 的内容进行 Hash 计算后的值，相当于签名，这样就能很方便地比较 RC 里的 Image 名字（以及其他信息）是否发生了变化。

`RunRollingUpdate` 执行逻辑的第一步：确定 New RC 对象及建立起 Old RC 到 New RC 的关联关系。下面我们以指定的 Image 参数进行 `rolling update` 的方式为例，看看代码是如何实现这段逻辑的。下面是相关源码：

```

if len(image) != 0 {
    keepOldName = len(args) == 1
    newName := findNewName(args, oldRc)
    if newRc, err = kubectl.LoadExistingNextReplicationController(client,
cmdNamespace, newName); err != nil {
        return err
    }
    if newRc != nil {
        fmt.Fprintf(out, "Found existing update in progress (%s), resuming.\n",
newRc.Name)
    } else {
        newRc, err = kubectl.CreateNewControllerFromCurrentController(client,
cmdNamespace, oldName, newName, image, deploymentKey)
        if err != nil {
            return err
        }
    }
    // Update the existing replication controller with pointers to the 'next'
controller
    // and adding the <deploymentKey> label if necessary to distinguish it from
the 'next' controller.
    oldHash, err := api.HashObject(oldRc, client.Codec)
    if err != nil {
        return err
    }
    oldRc, err = kubectl.UpdateExistingReplicationController(client, oldRc,
cmdNamespace, newRc.Name, deploymentKey, oldHash, out)
    if err != nil {
        return err
    }
}
}

```

在代码里，`findNewName` 方法查询新 RC 的名字，如果在命令行参数中没有提供新 RC 的名字，则从 Old RC 中根据 `kubectl.kubernetes.io/next-controller-id` 这个 `Annotation` 标签找新 RC 的名字并返回，如果新 RC 存在则继续使用，否则调用 `CreateNewControllerFromCurrentController`

方法创建一个新 RC，在新 RC 的创建过程中设定 deploymentKey 的值为自己的 Hash 签名，方法源码如下：

```
func CreateNewControllerFromCurrentController(c *client.Client, namespace, oldName,
newName, image, deploymentKey string) (*api.ReplicationController, error) {
    // load the old RC into the "new" RC
    newRc, err := c.ReplicationControllers(namespace).Get(oldName)
    if err != nil {
        return nil, err
    }
    if len(newRc.Spec.Template.Spec.Containers) > 1 {
        // TODO: support multi-container image update.
        return nil, goerrors.New("Image update is not supported for multi-container
pods")
    }
    if len(newRc.Spec.Template.Spec.Containers) == 0 {
        return nil, goerrors.New(fmt.Sprintf("Pod has no containers! (%v) ", newRc))
    }
    newRc.Spec.Template.Spec.Containers[0].Image = image
    newHash, err := api.HashObject(newRc, c.Codec)
    if err != nil {
        return nil, err
    }
    if len(newName) == 0 {
        newName = fmt.Sprintf("%s-%s", newRc.Name, newHash)
    }
    newRc.Name = newName
    newRc.Spec.Selector[deploymentKey] = newHash
    newRc.Spec.Template.Labels[deploymentKey] = newHash
    // Clear resource version after hashing so that identical updates get different
hashes.
    newRc.ResourceVersion = ""
    return newRc, nil
}
```

在 Image rolling update 的流程中确定新的 RC 以后，调用 UpdateExistingReplicationController 方法，将旧 RC 的 kubectl.kubernetes.io/next-controller-id 设置为新 RC 的名字，并且判断旧 RC 是否需要设置或更新 deploymentKey，具体代码如下：

```
func UpdateExistingReplicationController(c client.Interface, oldRc *api.
ReplicationController, namespace, newName, deploymentKey, deploymentValue string,
out io.Writer) (*api.ReplicationController, error) {
    SetNextControllerAnnotation(oldRc, newName)
    if _, found := oldRc.Spec.Selector[deploymentKey]; !found {
        return AddDeploymentKeyToReplicationController(oldRc, c, deploymentKey,
deploymentValue, namespace, out)
```

```

    } else {
        // If we didn't need to update the controller for the deployment key, we still
        need to write
        // the "next" controller.
        return c.ReplicationControllers(namespace).Update(oldRc)
    }
}

```

通过上面的逻辑，新 RC 被确定并且旧 RC 到新 RC 的关联关系也被建立好了，接下来如果 dry-run 参数为 true，则仅仅打印新旧 RC 的信息然后返回。如果是正常的 rolling update 动作，则创建一个 kubectl.RollingUpdater 对象来执行具体任务，任务的参数则放在 kubectl.RollingUpdaterConfig 中，相关源码如下：

```

updateCleanupPolicy := kubectl.DeleteRollingUpdateCleanupPolicy
if keepOldName {
    updateCleanupPolicy = kubectl.RenameRollingUpdateCleanupPolicy
}
config := &kubectl.RollingUpdaterConfig{
    Out:          out,
    OldRc:        oldRc,
    NewRc:        newRc,
    UpdatePeriod: period,
    Interval:     interval,
    Timeout:      timeout,
    CleanupPolicy: updateCleanupPolicy,
}

```



其中 out 是输出流（屏幕输出）；UpdatePeriod 是执行 rolling update 动作的间隔时间；Interval 与 Timeout 组合使用，前者是每次拉取 polling controller 状态的间隔时间，而后者则是对应的（HTTP REST 调用）超时时间。CleanupPolicy 确定升级结束后的善后策略，比如 DeleteRollingUpdateCleanupPolicy 表示删除旧的 RC，而 RenameRollingUpdateCleanupPolicy 则表示保持 RC 的名字不变（改变新 RC 的名字）。

RollingUpdater 的 Update 方法是 rolling update 的核心，它以上述 config 对象作为参数，其核心流程是每次让新 RC 的 Pod 副本数量加 1，同时旧 RC 的 Pod 副本数量减 1，直到新 RC 的 Pod 副本数量达到预期值同时旧 RC 的 Pod 副本数量变为零为止，在这个过程中由于新旧 RC 的 Pod 副本数量一直在变动，所以需要一个地方记录最初不变的那个 Pod 副本数量，这里就是 RC 的 Annotation 标签——kubectl.kubernetes.io/desired-replicas。

下面这段源码就是“贴标签”的过程：

```

fmt.Fprintf(out, "Creating %s\n", newName)
if newRc.ObjectMeta.Annotations == nil {
    newRc.ObjectMeta.Annotations = map[string]string{}
}

```

```

    newRc.ObjectMeta.Annotations[desiredReplicasAnnotation] = fmt.Sprintf
    ("%d", desired)
    newRc.ObjectMeta.Annotations[sourceIdAnnotation] = sourceId
    newRc.Spec.Replicas = 0
    newRc, err = r.c.CreateReplicationController(r.ns, n

```

下面这段源码便是“江山代有才人出，一代新人换旧人”的生动画面：

```

for newRc.Spec.Replicas < desired && oldRc.Spec.Replicas != 0 {
    newRc.Spec.Replicas += 1
    oldRc.Spec.Replicas -= 1
    fmt.Printf("At beginning of loop: %s replicas: %d, %s replicas: %d\n",
        oldName, oldRc.Spec.Replicas,
        newName, newRc.Spec.Replicas)
    fmt.Fprintf(out, "Updating %s replicas: %d, %s replicas: %d\n",
        oldName, oldRc.Spec.Replicas,
        newName, newRc.Spec.Replicas)
    newRc, err = r.scaleAndWait(newRc, retry, waitForReplicas)
    if err != nil {
        return err
    }
    time.Sleep(updatePeriod)
    oldRc, err = r.scaleAndWait(oldRc, retry, waitForReplicas)
    if err != nil {
        return err
    }
    fmt.Printf("At end of loop: %s replicas: %d, %s replicas: %d\n",
        oldName, oldRc.Spec.Replicas,
        newName, newRc.Spec.Replicas)
}
// delete remaining replicas on oldRc
if oldRc.Spec.Replicas != 0 {
    fmt.Fprintf(out, "Stopping %s replicas: %d -> %d\n",
        oldName, oldRc.Spec.Replicas, 0)
    oldRc.Spec.Replicas = 0
    oldRc, err = r.scaleAndWait(oldRc, retry, waitForReplicas)
    if err != nil {
        return err
    }
}
// add remaining replicas on newRc
if newRc.Spec.Replicas != desired {
    fmt.Fprintf(out, "Scaling %s replicas: %d -> %d\n",
        newName, newRc.Spec.Replicas, desired)
    newRc.Spec.Replicas = desired
    newRc, err = r.scaleAndWait(newRc, retry, waitForReplicas)
    if err != nil {
        return err
    }
}

```

```
    }  
}
```

上述方法里的 `scaleAndWait` 方法调用了 `kubectl.ReplicationControllerScaler` 的 `Scale` 方法，`Scale` 方法先通过 Rest API 调用 Kubernetes API Server 更新 RC 的 Pod 副本数量，然后循环拉取 RC 信息，直到超时或者 RC 同步状态完成。下面是判断 RC 同步状态是否完成的函数，来自 `client` 包 (`pkg/client/conditions.go`)。

```
func ControllerHasDesiredReplicas(c Interface, controller *api.ReplicationController)  
wait.ConditionFunc {  
    desiredGeneration := controller.Generation  
    return func() (bool, error) {  
        ctrl, err := c.ReplicationControllers(controller.Namespace).Get  
(controller.Name)  
        if err != nil {  
            return false, err  
        }  
        return ctrl.Status.ObservedGeneration >= desiredGeneration &&  
ctrl.Status.Replicas == ctrl.Spec.Replicas, nil  
    }  
}
```

`rolling-update` 是 `kubectl` 所有命令中最为复杂的一个，从它的功能和流程来看，完全可以被当作一个 Job 并放到 `kube-controller-manager` 上实现，客户端仅仅发起 Job 的创建及 Job 状态查看等命令即可，未来 Kubernetes 的版本是否会这样重构，我们拭目以待。

后记

Kubernetes 无疑是容器化技术时代最好的分布式系统架构，但是目前它还没有一款很好的图形化管理工具，基本上是命令行操作，因此不容易入门。另外，在系统运行过程中，我们难以直观了解当前服务的分布情况及资源的使用情况，日志也不完善，难以快速追踪和排查故障，因此，我们发起了一个名为 **Ku8eye** 的开源项目，这是借鉴了 **OpenStack Horizon**、**Cloudera Manager** 等知名软件的设计思想的一款国产开源软件，目标是成为 **Kubernetes** 的姊妹开源项目。

Ku8eye 作为 **Kubernetes** 的一站式管理工具，具备如下关键特性。

- ◎ 图形化一键安装和部署多节点 **Kubernetes** 集群。这是安装、部署谷歌 **Kubernetes** 集群的最快、最佳方式，其安装流程会参考当前的系统环境，提供默认优化的集群安装参数，实现最佳部署。
- ◎ 支持多角色、多租户的 Portal 管理界面。通过一个集中化的 Portal 界面，运营团队可以很方便地调整集群配置及管理集群资源，实现跨部门的角色、用户及多租户管理，通过自助服务可以很容易完成 **Kubernetes** 集群的运维管理工作。
- ◎ 制定了 **Kubernetes** 应用的程序发布包标准（**ku8package**），并提供了一款向导工具，使得专门为 **Kubernetes** 设计的应用能够很容易地从本地环境发布到公有云和其他环境中；并且提供了 **Kubernetes** 应用的可视化构建工具，实现了 **Kubernetes Service**、**RC**、**Pod** 及其他资源的可视化构建和管理功能。
- ◎ 可定制化的监控和告警系统。**Ku8eye** 内建了很多系统健康检查工具来检测、发现异常并触发告警事件，不仅可以监控集群中的所有节点和组件（包括 **Docker** 与 **Kubernetes**），还可以很容易地监控业务应用的性能；并且提供了一个强大的 **Dashboard**，用来生成各种复杂的监控图表以展示历史信息，还可用来自定义相关监控指标的告警阀值。
- ◎ 具备综合的全面的故障排查能力。**Ku8eye** 提供了集中化的唯一日志管理工具，日志系统从集群中的各个节点拉取日志并做聚合分析，拉取的日志包括系统日志和用户程序日志；并且提供了全文检索能力以方便故障分析和问题排查，检索的信息包括相关告警信息，而历史视图和相关的度量数据则告诉我们什么时候发生了什么事情，有助于

快速了解相关时间内系统的行为特征。

- ◎ 实现了 Docker 与 Kubernetes 项目的持续集成功能。Ku8eye 提供了一款可视化工具，用来驱动持续集成的整个流程，包括创建新的 Docker 镜像，Push 镜像到私有仓库，创建 Kubernetes 测试环境进行测试，以及最终滚动升级到生产环境中的各个主要环节。

Ku8eye 的 GitHub 地址为 <https://github.com/bestcloud>，Ku8eye 目前所用到的技术包括 Java Web、Ansible 脚本，未来可能涉及 Python 脚本及 Android 开发等。截至本书出版时，Ku8eye 已有 10 多名团队成员。如果您有兴趣，可在学完本书后加入本项目 QQ 群（Kubernetes 中国）：285431657。

