

Kubernetes

权威指南

龚正 吴治辉 叶伙荣
张龙春 闫健勇 / 编者

从Docker到Kubernetes
实践全接触



下一代分布式架构的王者

Kubernetes: The Definitive Guide



中国工信出版集团



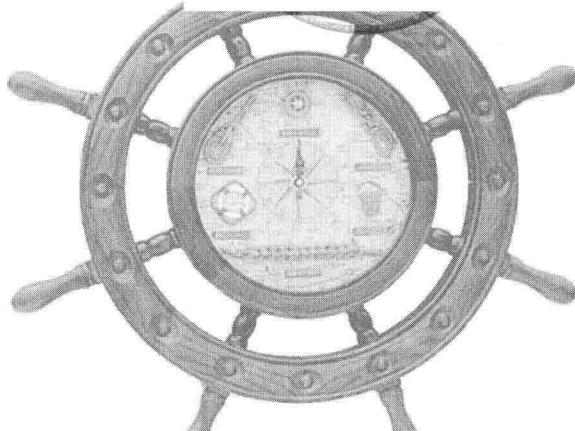
电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Kubernetes

权威指南

龚正 吴治辉 叶伙荣 /
张龙春 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、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、实战开发、运维、高级案例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，有很好的可借鉴性。

无论对于软件工程师、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. —北京：电子工业出版社，2016.1
ISBN 978-7-121-27639-2

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（2015）第 281677 号

策划编辑：张国霞

责任编辑：徐津平

印 刷：北京京师印务有限公司

装 订：北京京师印务有限公司

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：25.75 字数：580 千字

版 次：2016 年 1 月第 1 版

印 次：2016 年 1 月第 1 次印刷

印 数：4000 册 定价：89.00 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。

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。

服务热线：(010) 88258888。

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,《Kubernetes 权威指南——从 Docker 到 Kubernetes 实践全接触》终于与我们大家见面了。我有幸做为首批读者,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。Docker 的快速发展打败了谷歌曾经名噪一时的开源容器技术 Imctfy，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，不仅为当年失败的 Imctfy 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 虚机一样的平等公民。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。

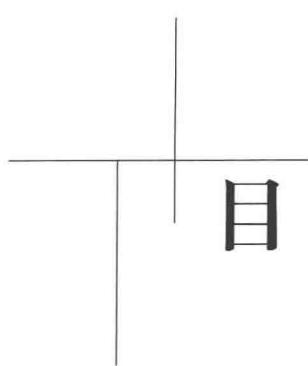
谷歌凭借着几十年大规模容器使用的丰富经验，步步为营，先是祭出 Kubernetes 这个神器，

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四百多页的 Kubernetes 权威指南。本书遵循从入门到精通的学习路线，全书共分为六大部分，涵盖了入门、高级案例、架构、原理、开发指南、运维及源码分析等内容，内容详实、图文并茂，几乎囊括了 Kubernetes 1.0 的方方面面，无论对于软件工程师、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	Kubernetes 是什么	1
1.2	为什么要用 Kubernetes	4
1.3	从一个不简单的 Hello World 例子说起	5
1.3.1	创建 redis-master Pod 和服务	7
1.3.2	创建 redis-slave Pod 和服务	10
1.3.3	创建 frontend Pod 和服务	12
1.3.4	通过浏览器访问网页	15
1.4	Kubernetes 基本概念和术语	16
1.4.1	Node (节点)	16
1.4.2	Pod	18
1.4.3	Label (标签)	20
1.4.4	Replication Controller (RC)	24
1.4.5	Service (服务)	26
1.4.6	Volume (存储卷)	30
1.4.7	Namespace (命名空间)	34
1.4.8	Annotation (注解)	35

1.4.9 小结	36
1.5 Kubernetes 总体架构	36
1.6 Kubernetes 安装与配置	38
1.6.1 安装 Kubernetes	38
1.6.2 配置和启动 Kubernetes 服务	39
1.6.3 Kubernetes 的版本升级	46
1.6.4 内网中的 Kubernetes 相关配置	46
1.6.5 Kubernetes 对 Docker 镜像的要求——启动命令前台执行	48

第 2 章 Kubernetes 核心原理 49

2.1 Kubernetes API Server 分析	49
2.1.1 如何访问 Kubernetes API	49
2.1.2 通过 API Server 访问 Node、Pod 和 Service	52
2.1.3 集群功能模块之间的通信	55
2.2 调度控制原理	56
2.2.1 Replication Controller	57
2.2.2 Node Controller	60
2.2.3 ResourceQuota Controller	62
2.2.4 Namespace Controller	64
2.2.5 ServiceAccount Controller 与 Token Controller	64
2.2.6 Service Controller 与 Endpoint Controller	65
2.2.7 Kubernetes Scheduler	71
2.3 Kubelet 运行机制分析	75
2.3.1 节点管理	75
2.3.2 Pod 管理	76
2.3.3 容器健康检查	77
2.3.4 cAdvisor 资源监控	78

2.4 安全机制的原理.....	80
2.4.1 Authentication 认证.....	80
2.4.2 Authorization 授权.....	83
2.4.3 Admission Control 准入控制	84
2.4.4 Secret 私密凭据.....	88
2.4.5 Service Account	92
2.5 网络原理	95
2.5.1 Kubernetes 网络模型.....	95
2.5.2 Docker 的网络基础.....	97
2.5.3 Docker 的网络实现	109
2.5.4 Kubernetes 的网络实现.....	117
2.5.5 开源的网络组件.....	127
2.5.6 Kubernetes 网络试验.....	131

第 3 章 Kubernetes 开发指南

145

3.1 REST 简述	145
3.2 Kubernetes API 详解	147
3.2.1 Kubernetes API 概述	147
3.2.2 API 版本	152
3.2.3 API 详细说明	152
3.2.4 API 响应说明	154
3.3 使用 Java 程序访问 Kubernetes API	156
3.3.1 Jersey.....	156
3.3.2 Fabric8	168
3.3.3 使用说明.....	169

第4章 Kubernetes 运维指南

191

4.1	Kubernetes 核心服务配置详解	191
4.1.1	基础公共配置参数	191
4.1.2	kube-apiserver	192
4.1.3	kube-controller-manager	195
4.1.4	kube-scheduler	196
4.1.5	Kubelet	197
4.1.6	kube-proxy	199
4.2	关键对象定义文件详解	200
4.2.1	Pod 定义文件详解	200
4.2.2	RC 定义文件详解	203
4.2.3	Service 定义文件详解	204
4.3	常用运维技巧集锦	206
4.3.1	Node 的隔离和恢复	206
4.3.2	Node 的扩容	207
4.3.3	Pod 动态扩容和缩放	208
4.3.4	更新资源对象的 Label	208
4.3.5	将 Pod 调度到指定的 Node	209
4.3.6	应用的滚动升级	210
4.3.7	Kubernetes 集群高可用方案	213
4.4	资源配额管理	217
4.4.1	指定容器配额	217
4.4.2	全局默认配额	218
4.4.3	多租户配额管理	221
4.5	Kubernetes 网络配置方案详解	223
4.5.1	直接路由方案	224

4.5.2 使用 flannel 叠加网络	226
4.5.3 使用 Open vSwitch	228
4.6 Kubernetes 集群监控	232
4.6.1 使用 kube-ui 查看集群运行状态	232
4.6.2 使用 cAdvisor 查看容器运行状态	236
4.7 Trouble Shooting 指导	241
4.7.1 对象的 Event 事件	242
4.7.2 容器日志	243
4.7.3 Kubernetes 系统日志	244
4.7.4 常见问题	246
4.7.5 寻求帮助	249
第 5 章 Kubernetes 高级案例进阶	250
5.1 Kubernetes DNS 服务配置案例	250
5.1.1 skydns 配置文件	251
5.1.2 修改每个 Node 上的 Kubelet 启动参数	254
5.1.3 创建 skydns Pod 和服务	254
5.1.4 通过 DNS 查找 Service	255
5.1.5 DNS 服务的工作原理解析	256
5.2 Kubernetes 集群性能监控案例	257
5.2.1 配置 Kubernetes 集群的 ServiceAccount 和 Secret	258
5.2.2 部署 Heapster、InfluxDB、Grafana	261
5.2.3 查询 InfluxDB 数据库中的数据	265
5.2.4 Grafana 页面查看和操作	268
5.3 Cassandra 集群部署案例	269
5.3.1 自定义 SeedProvider	270
5.3.2 通过 Service 动态查找 Pod	271

5.3.3 Cassandra 集群新节点的自动添加.....	274
5.4 集群安全配置案例.....	275
5.4.1 双向认证配置.....	275
5.4.2 简单认证配置.....	279
5.5 不同工作组共享 Kubernetes 集群的案例	280
5.5.1 创建 namespace	281
5.5.2 定义 Context (运行环境)	281
5.5.3 设置工作组在特定 Context 环境中工作.....	282

第 6 章 Kubernetes 源码导读 285

6.1 Kubernetes 源码结构和编译步骤	285
6.2 kube-apiserver 进程源码分析.....	289
6.2.1 进程启动过程.....	289
6.2.2 关键代码分析.....	291
6.2.3 设计总结.....	306
6.3 kube-controller-manager 进程源码分析	310
6.3.1 进程启动过程.....	310
6.3.2 关键代码分析.....	313
6.3.3 设计总结.....	321
6.4 kube-scheduler 进程源码分析.....	323
6.4.1 进程启动过程.....	323
6.4.2 关键代码分析.....	328
6.4.3 设计总结.....	335
6.5 Kubelet 进程源码分析	337
6.5.1 进程启动过程.....	337
6.5.2 关键代码分析.....	342
6.5.3 设计总结.....	365

6.6 kube-proxy 进程源码分析	366
6.6.1 进程启动过程	367
6.6.2 关键代码分析	368
6.6.3 设计总结	383
6.7 Kubectl 进程源码分析	384
6.7.1 kubectl create 命令	385
6.7.2 rolling-upate 命令	389
后记	396

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。

首先，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。

其次，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置管理能力。同时，Kubernetes 还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 my-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 就贴上 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 1.3 节介绍的 Hello World 为例，采用 RC 的方式，只要为 frontend 创建一个 3 副本的 RC，为 redis-master 创建一个单副本的 RC（这里单副本 RC 的意义就留给你来思考了），为 redis-slaver 创建一个 2 副本的 RC，总共 3 个文件，就分分钟完成整个集群的搭建过程了，是不是很有趣？

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 12315 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 从一个不简单的 Hello World 例子说起

典型的 Hello World 例子是在屏幕终端输出一句话“Hello World”，而这里的 Hello World 例子是一个 Web 留言板应用，并且是一个基于 PHP+Redis 的两层分布式架构的 Web 应用，前端 PHP Web 网站通过访问后端 Redis 数据库来完成用户留言的查询和添加等功能，更重要的是这个传统的经典案例部署在 Kubernetes 集群中，具备 Redis 读写分离能力。本章将通过这个案例带你一起走进 Kubernetes 的精彩世界。

留言板网页界面很简单，如图 1.1 所示，首页将显示访客的留言，留言内容是从 Redis 中查询得到的，首页提供一个文本输入框允许访客添加留言，添加的留言将被写入 Redis 中。

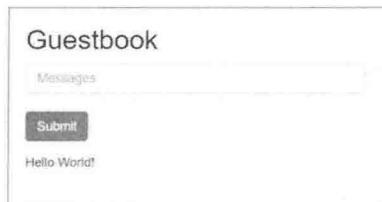


图 1.1 留言板网页界面

留言板的系统部署架构如图 1.2 所示。为了实现读写分离，在 Redis 层采用了一个 Master 与两个 Slave 的高可用集群模式进行部署，其中 Master 实例用于前端写操作（添加留言），而两个 Slave 实例则用于前端读操作（读取留言）。PHP 的 Web 层同样启动 3 个实例组成集群，实

现客户端（例如浏览器）对网站访问的负载均衡。

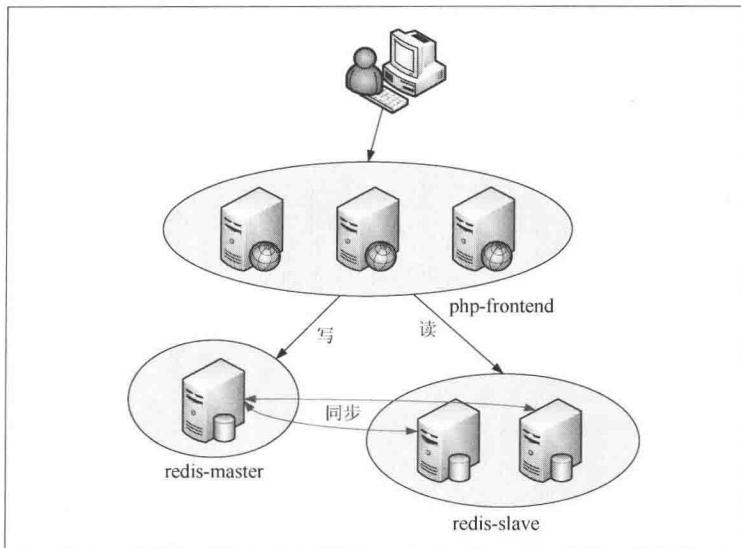


图 1.2 留言板的系统部署架构图

接下来，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，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（更为详细的安装步骤会在后面给出）。

(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。接下来我们需要下载案例中要用到的以下 3 个 Docker 镜像。

- ◎ **redis-master**: 用于前端 Web 应用进行“写”留言的操作，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”留言的操作，并与 **redis-master** 的数据保持同步。
- ◎ **guestbook-php-frontend**: PHP Web 服务，在网页上展示留言内容，也提供一个文本输入框供访客添加留言。

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

如图 1.3 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，最终完成整个例子。

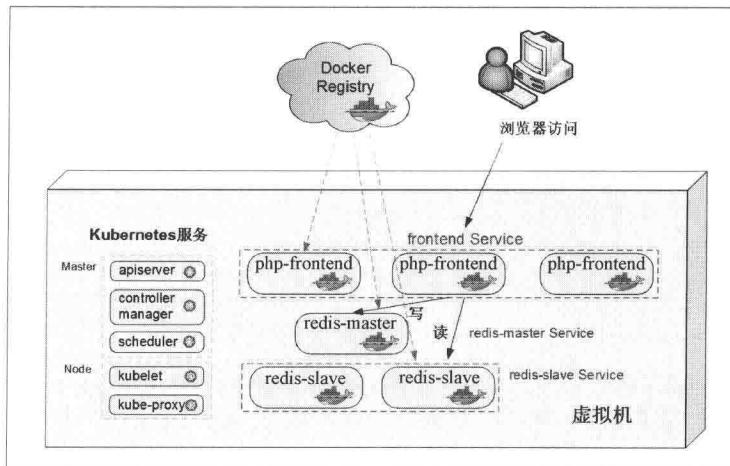


图 1.3 Kubernetes 部署架构图

1.3.1 创建 redis-master Pod 和服务

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，这两种方式最终的结果都一样，这里我们采用后一种思路。

首先为 redis-master 服务创建一个名为 redis-master 的 RC 定义文件:redis-master-controller.yaml。下面给出了该文件的完整内容:

```
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 表示只能运行一个（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
replicationcontrollers/redis-master
```

系统提示“replicationcontrollers/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
CONTROLLER   CONTAINER(S)   IMAGE(S)           SELECTOR           REPLICAS
redis-master  master        kubeguide/redis-master  name=redis-master  1
```

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-xxx` 的 Pod 实例，这是 Kubernetes 根据 `redis-master` 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 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。

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

提供 Redis 服务的 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）——`redis-master` 的定义文件（文件名为 `redis-master-service.yaml`），完整内容如下：

```
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` 确定了哪些 Pod 对应到本服务，这里的定义表明拥有 `redis-master` 标签的 Pod 属于 `redis-master` 服务。另外，`ports` 部分中的 `targetPort` 属性用来确定提供该服务的容器所暴露（EXPOSE）的端口号，即具体的服务进程在容器内的 `targetPort` 上提供服务，而 `port` 属性则定义了 Service 的虚端口。

运行 Kubectl，创建 service：

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

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

```
$ kubectl get services
NAME           LABELS            SELECTOR           IP(S)        PORT(S)
redis-master   name=redis-master   name=redis-master 10.254.208.57  6379/TCP
```

注意到 `redis-master` 服务被分配了一个值为 `10.254.208.57` 的 IP 地址（虚拟 IP），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` 来访问它了。在本例中，将要创建的 `redis-slave` 和 `frontend` 两组 Pods 都将通过 `10.254.208.57:6379` 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=10.254.144.74  
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。

1.3.2 创建 redis-slave Pod 和服务

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 所对应的 Redis 进程进行数据同步，3 个 Redis 实例组成一个具备读写分离能力的 Redis 集群。留言板的 PHP 程序通过访问 redis-slave 服务来获取已保存的留言数据。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（文件名为 redis-slave-controller.yaml）。下面给出了该文件的完整内容：

```
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

```

运行 kubectl create 命令：

```
$ kubectl create -f redis-slave-controller.yaml
replicationcontrollers/redis-slave
```

运行 kubectl get 命令查看 RC：

```
$ kubectl get rc
CONTROLLER      CONTAINER(S)        IMAGE (S)          SELECTOR          REPLICAS
redis-master    master             kubeguide/redis-master   name=redis-master  1
redis-slave     slave              kubeguide/guestbook-redis-slave  name=redis-slave  2
```

查看 RC 创建的 Pods，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
```

为了实现 Redis 集群的主从数据同步，redis-slave 需要知道 redis-master 的地址，所以在 redis-slave 镜像的启动命令 /run.sh 中，我们可以输入如下内容：

```
redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST} 6379
```

由于在创建 redis-slave Pod 时，系统自动在容器内部生成与 redis-master Service 相关的环境变量，所以 redis-slave 应用程序能够直接使用环境变量 REDIS_MASTER_SERVICE_HOST 来获取 redis-master 服务的 IP 地址。

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在后续新建的 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           LABELS             SELECTOR          IP(S)        PORT(S)  
redis-master   name=redis-master   name=redis-master  10.254.208.57  6379/TCP  
redis-slave    name=redis-slave    name=redis-slave   10.254.78.102  6379/TCP
```

1.3.3 创建 frontend Pod 和服务

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
```

我们注意到 Pod 里提供的容器镜像为 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();
} ?>

```

这段代码很简单，如果是一个 set 请求（提交留言），则会创建一个连接 redis_master 服务的 Redis 客户端保存数据，其中 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）；否则是一个查询请求，对连接 redis_slave 服务进行查询操作。

运行 kubectl create 命令创建 RC：

```
$ kubectl create -f frontend-controller.yaml
replicationcontrollers/frontend
```

查看已创建的 RC：

```
$ kubectl get rc
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS
redis-master	master	kubeguide/redis-masternode=redis-master		1
redis-slave	slave	kubeguide/redis-slave name=redis-slave		2
frontend	php-redis	kubeguide/guestbook-php-frontend	name=frontend	3

再查看生成的 Pod:

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 的端口号定义有范围限制，默认为 30000~32767，如果配置为范围外的其他端口号，则创建 Service 将会失败。

运行 Kubectl 创建 Service:

```
$ kubectl create -f frontend-service.yaml
services/frontend
```

通过 Kubectl 查看创建的 Service:

NAME	LABELS	SELECTOR	IP(S)	PORT(S)
redis-master	name=redis-master	name=redis-master	10.254.208.57	6379/TCP

redis-slave	name=redis-slave	name=redis-slave	10.254.78.102	6379/TCP
frontend	name=frontend	name=frontend	10.254.167.153	80/TCP

1.3.4 通过浏览器访问网页

经过上面的三个步骤，我们终于成功实现了留言板系统在 Kubernetes 上的部署工作，现在一起来见证成果吧，在你的笔记本上打开浏览器，输入下面的 URL：<http://虚拟机 IP:30001>。

如果看到如图 1.4 所示的网页，并且看到网页上有一条留言——“Hello World!”，那么恭喜你，之前的努力没有白费，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……

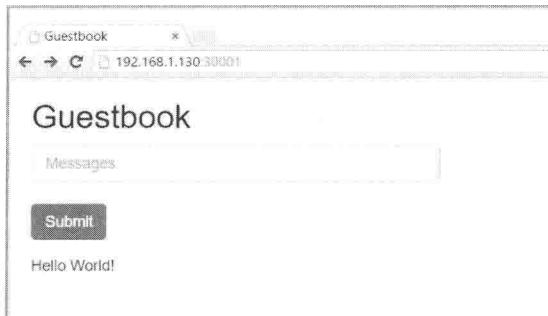


图 1.4 通过浏览器访问留言板网页

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 1.5 所示。



图 1.5 在留言板网页添加新的留言

至此，我们终于完成了 Kubernetes 上的 Hello World 例子，这个例子并不简单，但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，这不能不让我们惊诧于它的创新和强大。

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里的 Hello World 例子，比如：

- ◎ 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

在 Kubernetes 中，Node、Pod、Replication Controller、Service 等概念都可以看作一种资源对象，通过 Kubernetes 提供的 Kubectl 工具或者 API 调用进行操作，并保存在 etcd 中。

1.4.1 Node（节点）

Node（节点）是 Kubernetes 集群中相对于 Master 而言的工作主机，在较早的版本中也被称为 Minion。Node 可以是一台物理主机，也可以是一台虚拟机（VM）。在每个 Node 上运行用于启动和管理 Pod 的服务——Kubelet，并能够被 Master 管理。在 Node 上运行的服务进程包括 Kubelet、kube-proxy 和 docker daemon。

Node 的信息如下。

- ◎ Node 地址：主机的 IP 地址，或者 Node ID。
- ◎ Node 运行状态：包括 Pending、Running、Terminated 三种状态。
- ◎ Node Condition（条件）：描述 Running 状态 Node 的运行条件，目前只有一种条件——Ready。Ready 表示 Node 处于健康状态，可以接收从 Master 发来的创建 Pod 的指令。
- ◎ Node 系统容量：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。
- ◎ 其他：Node 的其他信息，包括实例的内核版本号、Kubernetes 版本号、Docker 版本号、操作系统名称等。

我们可以通过 kubectl describe node <node_name> 来查看 Node 的详细信息，例如：

```
$ kubectl describe node kubernetes-minion1
Name: kubernetes-minion1
Labels: kubernetes.io/hostname=kubernetes-minion1
CreationTimestamp: Tue, 04 Aug 2015 14:34:22 +0800
Conditions:
  Type    Status  LastHeartbeatTime     LastTransitionTime   Reason     Message
  Ready      True   Thu, 13 Aug 2015 12:12:03 +0800   Tue, 11 Aug 2015
09:37:17 +0800           kubelet is posting ready status
Addresses: 192.168.1.129
Capacity:
  cpu:        2
  memory:    1870516Ki
  pods:       40
Version:
  Kernel Version:      3.10.0-229.el7.x86_64
  OS Image:            Red Hat Enterprise Linux Server 7.1 (Maipo)
  Container Runtime Version: docker://1.6.2.el7
  Kubelet Version:     v1.0.0
  Kube-Proxy Version:  v1.0.0
ExternalID:
Pods:
  Namespace          Name
  default            frontend-o8bg4
  default            frontend-ulxxr
  default            frontend-z65iu
  default            redis-master-6okig
  default            redis-slave-4na2n
  default            redis-slave-92u3k
No events.
```

1. Node 的管理

Node 通常是物理机、虚拟机或者云服务商提供的资源，并不是由 Kubernetes 创建的。我们说 Kubernetes 创建一个 Node，仅仅表示 Kubernetes 在系统内部创建了一个 Node 对象，创建后即会对其进行一系列健康检查，包括是否可以连通、服务是否正确启动、是否可以创建 Pod 等。如果检查未能通过，则该 Node 将会在集群中被标记为不可用（Not Ready）。

2. 使用 Node Controller 对 Node 进行管理

Node Controller 是 Kubernetes Master 中的一个组件，用于管理 Node 对象。它的两个主要功能包括：集群范围内的 Node 信息同步，以及单个 Node 的生命周期管理。

Node 信息同步可以通过 kube-controller-manager 的启动参数--node-sync-period 设置同步的

时间周期。

3. Node 的自注册

当 Kubelet 的--register-node 参数被设置为 true (默认值即为 true) 时, Kubelet 会向 apiserver 注册自己。这也是 Kubernetes 推荐的 Node 管理方式。

Kubelet 进行自注册的启动参数如下。

- ◎ --apiservers=: apiserver 的地址;
- ◎ --kubeconfig=: 登录 apiserver 所需凭据/证书的目录;
- ◎ --cloud_provider=: 云服务商地址, 用于获取自身的 metadata;
- ◎ --register-node=: 设置为 true 表示自动注册到 apiserver。

4. 手动管理 Node

Kubernetes 集群管理员也可以手工创建和修改 Node 对象。当需要这样操作时, 先要将 Kubelet 启动参数中的--register-node 参数的值设置为 false。这样, 在 Node 上的 Kubelet 就不会把自己注册到 apiserver 中去。

另外, Kubernetes 提供了一种运行时加入或者隔离某些 Node 的方法。具体的操作请参考第 4 章。

1.4.2 Pod

Pod 是 Kubernetes 的最基本的操作单元, 包含一个或多个紧密相关的容器, 类似于豌豆荚的概念。一个 Pod 可以被一个容器化的环境看作应用层的“逻辑宿主机”(Logical Host)。一个 Pod 中的多个容器应用通常是紧耦合的。Pod 在 Node 上被创建、启动或者销毁。

为什么 Kubernetes 使用 Pod 在容器之上再封装一层呢? 一个很重要的原因是, Docker 容器之间的通信受到 Docker 网络机制的限制。在 Docker 的世界中, 一个容器需要通过 link 方式才能访问另一个容器提供的服务 (端口)。大量容器之间的 link 将是一件非常繁重的工作。通过 Pod 的概念将多个容器组合在一个虚拟的“主机”内, 可以实现容器之间仅需通过 Localhost 就能相互通信了。

Pod、容器与 Node 的关系如图 1.6 所示。

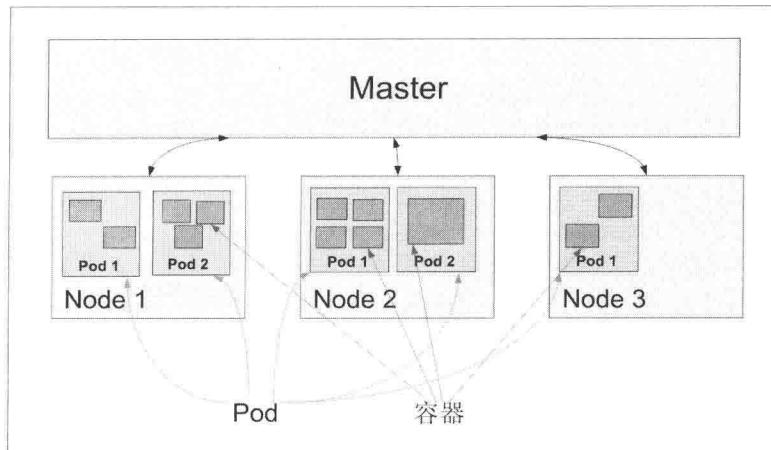


图 1.6 Pod、容器与 Node 的关系

一个 Pod 中的应用容器共享同一组资源，如下所述。

- ◎ PID 命名空间：Pod 中的不同应用程序可以看到其他应用程序的进程 ID。
- ◎ 网络命名空间：Pod 中的多个容器能够访问同一个 IP 和端口范围。
- ◎ IPC 命名空间：Pod 中的多个容器能够使用 SystemV IPC 或 POSIX 消息队列进行通信。
- ◎ UTS 命名空间：Pod 中的多个容器共享一个主机名。
- ◎ Volumes（共享存储卷）：Pod 中的各个容器可以访问在 Pod 级别定义的 Volumes。

1. 对 Pod 的定义

对 Pod 的定义通过 Yaml 或 Json 格式的配置文件来完成。下面的配置文件将定义一个名为 redis-slave 的 Pod，其中 kind 为 Pod。在 spec 中主要包含了对 Containers（容器）的定义，可以定义多个容器。

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-slave
  labels:
    name: redis-slave
spec:
  containers:
  - name: slave
    image: kubeguide/guestbook-redis-slave
    env:
    - name: GET_HOSTS_FROM
```

```

    value: env
  ports:
  - containerPort: 6379

```

Pod 的生命周期是通过 Replication Controller 来管理的。Pod 的生命周期过程包括：通过模板进行定义，然后分配到一个 Node 上运行，在 Pod 所含容器运行结束后 Pod 也结束。在整个过程中，Pod 处于以下 4 种状态之一，如图 1.7 所示。

- ◎ Pending：Pod 定义正确，提交到 Master，但其所包含的容器镜像还未完全创建。通常 Master 对 Pod 进行调度需要一些时间，之后 Node 对镜像进行下载也需要一些时间。
- ◎ Running：Pod 已被分配到某个 Node 上，且其包含的所有容器镜像都已经创建完成，并成功运行起来。
- ◎ Succeeded：Pod 中所有容器都成功结束，并且不会被重启，这是 Pod 的一种最终状态。
- ◎ Failed：Pod 中所有容器都结束了，但至少一个容器是以失败状态结束的，这也是 Pod 的一种最终状态。

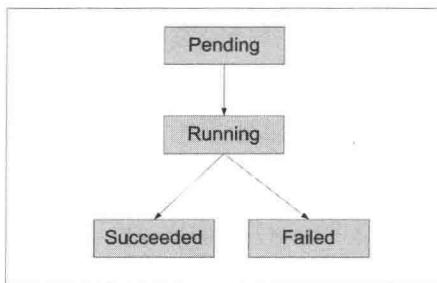


图 1.7 Pod 生命周期中的 4 种状态

Kubernetes 为 Pod 设计了一套独特的网络配置，包括：为每个 Pod 分配一个 IP 地址，使用 Pod 名作为容器间通信的主机名等。关于 Kubernetes 网络的设计原理将在第 2 章进行详细说明。

另外，不建议在 Kubernetes 的一个 Pod 内运行相同应用的多个实例。

1.4.3 Label（标签）

Label 是 Kubernetes 系统中的一个核心概念。Label 以 key/value 键值对的形式附加到各种对象上，如 Pod、Service、RC、Node 等。Label 定义了这些对象的可识别属性，用来对它们进行管理和选择。Label 可以在创建对象时附加到对象上，也可以在对象创建后通过 API 进行管理。

在为对象定义好 Label 后，其他对象就可以使用 Label Selector（选择器）来定义其作用的对象了。

Label Selector 的定义由多个逗号分隔的条件组成。

```
"labels": {
    "key1": "value1",
    "key2": "value2"
}
```

当前有两种 Label Selector：基于等式的（Equality-based）和基于集合的（Set-based），在使用时可以将多个 Label 进行组合来选择。

基于等式的 Label Selector 使用等式类的表达式来进行选择。

- ◎ name = redis-slave: 选择所有包含 Label 中 key= " name " 且 value= " redis-slave " 的对象。
- ◎ env != production: 选择所有包括 Label 中 key= " env " 且 value 不等于 " production " 的对象。

基于集合的 Label Selector 使用集合操作的表达式来进行选择。

- ◎ name in (redis-master, redis-slave): 选择所有包含 Label 中 key= " name " 且 value= " redis-master " 或 " redis-slave " 的对象。
- ◎ name not in (php-frontend): 选择所有包含 Label 中 key= " name " 且 value 不等于 " php-frontend " 的对象。

在某些对象需要对另一些对象进行选择时，可以将多个 Label Selector 进行组合，使用逗号 “,” 进行分隔即可。基于等式的 Label Selector 和基于集合的 Label Selector 可以任意组合。例如：

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

我们在使用 Label Selector 时，可以将其看作 SQL 查询语句中的 where 查询条件的语法。

例如，name=redis-slave 可以类比于 select * from <all_pods> where pod's name = 'redis-slave' 这样的 SQL 语句；name in (redis-master, redis-slave) 可以类比于 select * from <all_pods> where pod's name in (redis-master, redis-slave) 这样的 SQL 语句。而组合条件则相当于多条件的逻辑 AND 结果，例如 name=redis-slave,env!=production 类似于 select * from <all_pods> where pod's name = 'redis-slave' AND env <> 'production' 这样的 SQL 语句。

一般来说，我们会给一个 Pod（或其他对象）定义多个 Labels，以便于配置、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。通过对多个 Label 的设置，我们就可以“多维度”地对 Pod 或其他对象进行精细的管理。一些常用的 Label 示例如下：

- ◎ " release " : " stable " , " release " : " canary " ...
- ◎ " environment " : " dev " , " environment " : " qa " , " environment " : " production "

- ◎ "tier" : "frontend", "tier" : "backend", "tier" : "middleware"
- ◎ "partition" : "customerA", "partition" : "customerB" ...
- ◎ "track" : "daily", "track" : "weekly"

Replication Controller 通过 Label Selector 来选择要管理的 Pod。我们再看看 redis-slave RC 的定义。

在 RC 的定义中, Pod 部分的 template.metadata.labels 定义了 Pod 的 Label, 即 name=redis-slave。然后在 spec.selector 中指定 name=redis-slave, 表示将对所有包含该 Label 的 Pod 进行管理。

```
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: redis-slave
          ports:
            - containerPort: 6379
```

同样, 在 Service 的定义中, 也通过定义 spec.selector 为 name=redis-slave 来选择将哪些具有该 Label 的 Pod 加入其 Load Balance 的后端列表中去。这样, 当客户端访问请求到达该 Service 时, 系统就能够将请求转发到后端具有该 Label 的一个 Pod 上去。

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

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 定义了不同的取值。如图 1.8 所示，如果我们设置了“role=frontend”的 Selector，则会选取到 Node 1 和 Node 2 上的 Pod。

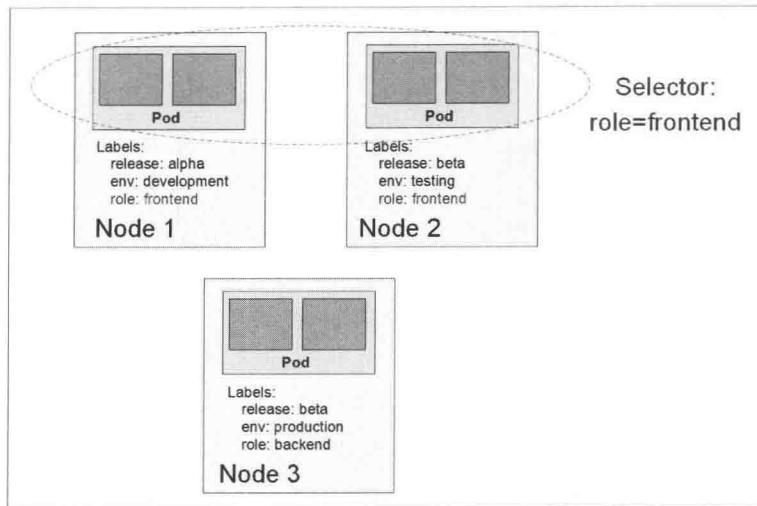


图 1.8 Label Selector 的作用范围 1

而设置“release=beta”的 Selector，则会选取到 Node 2 和 Node 3 上的 Pod，如图 1.9 所示。

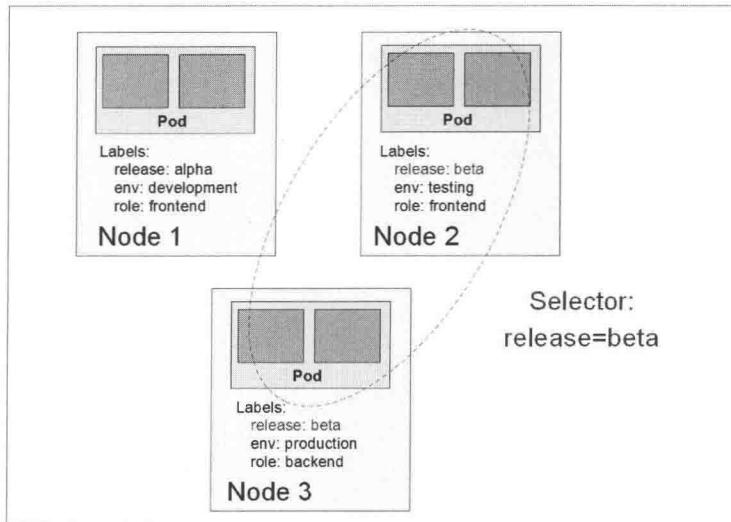


图 1.9 Label Selector 的作用范围 2

使用 Label 可以给对象创建多组标签，Service、RC 等组件则通过 Label Selector 来选择对象范围，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。

1.4.4 Replication Controller (RC)

之前已经对 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。

Replication Controller 是 Kubernetes 系统中的核心概念，用于定义 Pod 副本的数量。在 Master 内，Controller Manager 进程通过 RC 的定义来完成 Pod 的创建、监控、启停等操作。

根据 Replication Controller 的定义，Kubernetes 能够确保在任意时刻都能运行用户指定的 Pod “副本”（Replica）数量。如果有过多的 Pod 副本在运行，系统就会停掉一些 Pod；如果运行的 Pod 副本数量太少，系统就会再启动一些 Pod，总之，通过 RC 的定义，Kubernetes 总是保证集群中运行着用户期望的副本数量。

同时，Kubernetes 会对全部运行的 Pod 进行监控和管理，如果有需要（例如某个 Pod 停止运行），就会将 Pod 重启命令提交给 Node 上的某个程序来完成（如 Kubelet 或 Docker）。

可以说，通过对 Replication Controller 的使用，Kubernetes 实现了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）。

对 Replication Controller 的定义使用 Yaml 或 JSON 格式的配置文件来完成。以 redis-slave 为例，在配置文件中通过 spec.template 定义 Pod 的属性（这部分定义与 Pod 的定义是一致的），设置 spec.replicas=2 来定义 Pod 副本的数量。

```
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

```

通常，Kubernetes 集群中不止一个 Node，假设一个集群有 3 个 Node，根据 RC 的定义，系统将可能在其中的两个 Node 上创建 Pod。图 1.10 描述了在两个 Node 上创建 redis-slave Pod 的情形。

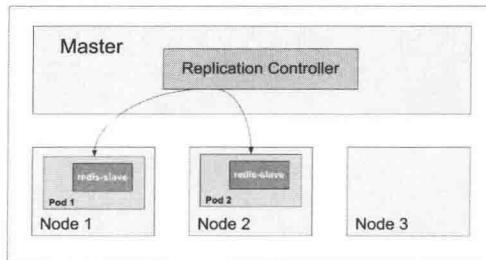


图 1.10 在两个 Node 上创建 redis-slave Pod

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

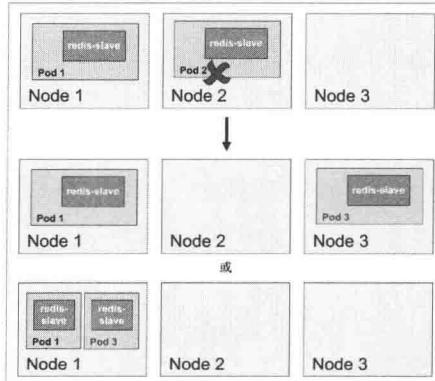


图 1.11 根据 RC 定义创建新的 Pod

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）。

Kubernetes 提供了 `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 1.12 所示。

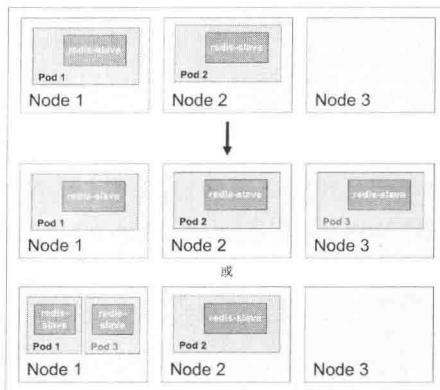


图 1.12 Scaling 的执行结果

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 值为 0，然后更新该 RC。另外，客户端工具 Kubectl 提供了 stop 和 delete 命令来完成一次性删除 RC 和 RC 控制的全部 Pod。

另外，通过修改 RC 可以实现应用的滚动升级（Rolling Update），具体的操作方法详见第 4 章。

1.4.5 Service（服务）

在 Kubernetes 的世界里，虽然每个 Pod 都会被分配一个单独的 IP 地址，但这个 IP 地址会随着 Pod 的销毁而消失。这就引出一个问题：如果有一组 Pod 组成一个集群来提供服务，那么如何来访问它们呢？

Kubernetes 的 Service（服务）就是用来解决这个问题的核心概念。

一个 Service 可以看作一组提供相同服务的 Pod 的对外访问接口。Service 作用于哪些 Pod 是通过 Label Selector 来定义的。

再看看上一节的例子，redis-slave Pod 运行了两个副本（replica），这两个 Pod 对于前端程序（frontend）来说没有区别，所以前端程序并不关心是哪个后端副本在提供服务。并且后端 redis-slave Pod 在发生变化时，前端也无须跟踪这些变化。“Service” 就是用来实现这种解耦的抽象概念。

1. 对 Service 的定义

对 Service 的定义同样使用 Yaml 或 JSON 格式的配置文件来完成。以 redis-slave 服务的定义为例：

```

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

```

通过该定义，Kubernetes 将会创建一个名为“redis-slave”的服务，并在 6379 端口上监听。spec.selector 的定义表示该 Service 将包含所有具有“name=redis-slave”Label 的 Pod。

在 Pod 正常启动后，系统将会根据 Service 的定义创建出与 Pod 对应的 Endpoint（端点）对象，以建立起 Service 与后端 Pod 的对应关系。随着 Pod 的创建、销毁，Endpoint 对象也将被更新。Endpoint 对象主要由 Pod 的 IP 地址和容器需要监听的端口号组成，通过 kubectl get endpoints 命令可以查看，显示为 IP:Port 的格式。

```

$ kubectl get endpoints
NAME           ENDPOINTS
redis-master   172.16.42.6:6379

```

2. Pod 的 IP 地址和 Service 的 Cluster IP 地址

Pod 的 IP 地址是 Docker Daemon 根据 docker0 网桥的 IP 地址段进行分配的，但 Service 的 Cluster IP 地址是 Kubernetes 系统中的虚拟 IP 地址，由系统动态分配。Service 的 Cluster IP 地址相对于 Pod 的 IP 地址来说相对稳定，Service 被创建时即被分配一个 IP 地址，在销毁该 Service 之前，这个 IP 地址都不会再变化了。而 Pod 在 Kubernetes 集群中生命周期较短，可能被 ReplicationController 销毁、再次创建，新创建的 Pod 将会被分配一个新的 IP 地址。

3. 外部访问 Service

由于 Service 对象在 Cluster IP Range 池中分配到的 IP 只能在内部访问，所以其他 Pod 都可以无障碍地访问到它。但如果这个 Service 作为前端服务，准备为集群外的客户端提供服务，我们就需要给这个服务提供公共 IP 了。

Kubernetes 支持两种对外提供服务的 Service 的 type 定义：NodePort 和 LoadBalancer。

1) NodePort

在定义 Service 时指定 spec.type=NodePort，并指定 spec.ports.nodePort 的值，系统就会在

Kubernetes 集群中的每个 Node 上打开一个主机上的真实端口号。这样，能够访问 Node 的客户端都就能通过这个端口号访问到内部的 Service 了。

以 php-frontend service 的定义为例，nodePort=80，这样，在每一个启动了该 php-frontend Pod 的 Node 节点上，都会打开 80 端口。

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30001
  selector:
    name: frontend
```

假设有 3 个 php-frontend Pod 运行在 3 个不同的 Node 上，客户端访问其中任意一个 Node 都可以访问到这个服务，如图 1.13 所示。



图 1.13 通过不同的物理机访问同一个服务

2) LoadBalancer

如果云服务商支持外接负载均衡器，则可以通过 spec.type=LoadBalancer 定义 Service，同时需要指定负载均衡器的 IP 地址。使用这种类型需要指定 Service 的 nodePort 和 clusterIP。例如：

```
apiVersion: v1
kind: Service
metadata: {
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "my-service"
```

```
        },
        "spec" : {
            "type" : "LoadBalancer",
            "clusterIP" : "10.0.171.239",
            "selector" : {
                "app" : "MyApp"
            },
            "ports" : [
                {
                    "protocol" : "TCP",
                    "port" : 80,
                    "targetPort" : 9376,
                    "nodePort" : 30061
                }
            ],
            "status" : {
                "loadBalancer" : {
                    "ingress" : [
                        {
                            "ip" : "146.148.47.155"
                        }
                    ]
                }
            }
        }
    }
```

在这个例子中，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。

之后，对该 Service 的访问请求将会通过 LoadBalancer 转发到后端 Pod 上去，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。

4. 多端口的服务

在很多情况下，一个服务都需要对外暴露多个端口号。在这种情况下，可以通过端口进行命名，使各 Endpoint 不会因重名而产生歧义。例如：

```
{  
    "kind": "Service",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "my-service"  
    },  
    "spec": {  
        "selector": {
```

```

        "app": "MyApp"
    },
    "ports": [
        {
            "name": "http",
            "protocol": "TCP",
            "port": 80,
            "targetPort": 9376
        },
        {
            "name": "https",
            "protocol": "TCP",
            "port": 443,
            "targetPort": 9377
        }
    ]
}
}
}

```

1.4.6 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 概念与 Docker 的 Volume 比较类似，但并不完全相同。Kubernetes 中的 Volume 与 Pod 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。另外，Kubernetes 支持多种类型的 Volume，并且一个 Pod 可以同时使用任意多个 Volume。

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。在同一个 Pod 中所有容器可以读和写 EmptyDir 中的相同文件。当 Pod 从 Node 上移除时，EmptyDir 中的数据也会永久删除。

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 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 EmptyDirs 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。

(2) **hostPath**: 在 Pod 上挂载宿主机上的文件或目录。

hostPath 通常可以用于：

- ◎ 容器应用程序生成的日志文件需要永久保存，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，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：

- ◎ 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。

以 redis-master 为例，使用宿主机的 /data 目录作为其容器内部挂载点 /data 的 volume。

在配置文件中，我们先在 Pod 的 spec 部分定义一个 volume，然后在 containers 中给容器定义 volumeMounts，name 为 volume 的名称。

```
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:
      volumes:
        - name: "persistent-storage"
          hostPath:
            path: "/data"
      containers:
        - name: master
          image: kubeguide/redis-master
          ports:
            - containerPort: 6379
          volumeMounts:
            - name: "persistent-storage"
              mountPath: "/data"
```

(3) gcePersistentDisk: 使用这种类型的 Volume 表示使用谷歌计算引擎 (Google Compute Engine, GCE) 上永久磁盘 (Persistent Disk, PD) 上的文件。与 EmptyDir 不同, PD 上的内容会永久保存, 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
```

在 Pod 定义中使用 gcePersistentDisk 示例:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: gcr.io/google_containers/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-pd
          name: test-volume
  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 Amazon 提供的 Amazon Web Services (AWS) 的 EBS Volume, 并可以挂载到 Pod 中去。需要注意的是, 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
```

在 Pod 定义中使用 gcePersistentDisk 示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: test-ebs
spec:
  containers:
    - image: gcr.io/google_containers/test-webserver
      name: test-container
    volumeMounts:
      - mountPath: /test-ebs
        name: test-volume
  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）提供的共享目录挂载到 Pod 中。在系统中需要一个运行中的 NFS 系统。

在 Pod 定义中使用 nfs 示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-web
spec:
  containers:
    - name: web
      image: nginx
    ports:
      - name: web
        containerPort: 80
    volumeMounts:
      # name must match the volume name below
      - name: nfs
        mountPath: "/usr/share/nginx/html"
  volumes:
    - name: nfs
      nfs:
        # 改为你的 NFS 服务器地址
        server: nfs-server.localhost
        path: "/"
```

(6) iscsi：使用 iSCSI 存储设备上的目录挂载到 Pod 中。

- (7) glusterfs: 使用开源 GlusterFS 网络文件系统的目录挂载到 Pod 中。
- (8) rbd: 使用 Linux 块设备共享存储 (Rados Block Device) 挂载到 Pod 中。
- (9) gitRepo: 通过挂载一个空目录，并从 GIT 库 clone 一个 git repository 以供 Pod 使用。
- (10) 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。
- (11) persistentVolumeClaim: 从 PV (PersistentVolume) 中申请所需的空间，PV 通常是一种网络存储，例如 GCEPersistentDisk、AWSElasticBlockStore、NFS、iSCSI 等。

1.4.7 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，通过将系统内部的对象“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 都将被系统创建到名为“default”的 Namespace 中。

用户可以根据需要创建新的 Namespace，通过如下 namespace-dev.yaml 文件进行创建：

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

再次查看系统中的 Namespace：

```
$ kubectl get namespaces
NAME      LABELS      STATUS
default   <none>     Active
development   <none>     Active
```

接着，在创建 Pod 时，可以指定 Pod 属于哪个 Namespace：

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

```

metadata:
  name: busybox
  namespace: development
spec:
  containers:
  - image: gcr.io/google_containers/busybox
    command:
      - sleep
      - "3600"
    name: busybox

```

在集群中，新创建的 Pod 将会属于“development”命名空间。

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，kubectl get 命令将仅显示属于“default”命名空间的对象。

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
```

使用 Namespace 来组织 Kubernetes 的各种对象，可以实现对用户的分组，即“多租户”管理。对不同的租户还可以进行单独的资源配置设置和管理，使得整个集群的资源配置非常灵活、方便。

关于多租户配额的详细配置方法请参考第 4 章中的详细介绍。

1.4.8 Annotation（注解）

Annotation 与 Label 类似，也使用 key/value 键值对的形式进行定义。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。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。

用 Annotation 来记录的信息包括：

- ② 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；
- ③ 日志库、监控库、分析库等资源库的地址信息；
- ④ 程序调试工具信息，例如工具名称、版本号等；
- ⑤ 团队的联系信息，例如电话号码、负责人名称、网址等。

1.4.9 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。

除了以上核心组件，在 Kubernetes 系统中还有许多可供配置的资源对象，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。

1.5 Kubernetes 总体架构

Kubernetes 集群由两类节点组成：Master 和 Node。在 Master 上运行 etcd、API Server、Controller Manager 和 Scheduler 四个组件，其中后三个组件构成了 Kubernetes 的总控中心，负责对集群中所有资源进行管控和调度。在每个 Node 上运行 Kubelet、Proxy 和 Docker Daemon 三个组件，负责对本节点上的 Pod 的生命周期进行管理，以及实现服务代理的功能。另外在所有节点上都可以运行 Kubectl 命令行工具，它提供了 Kubernetes 的集群管理工具集。图 1.14 描述了 Kubernetes 的系统架构。

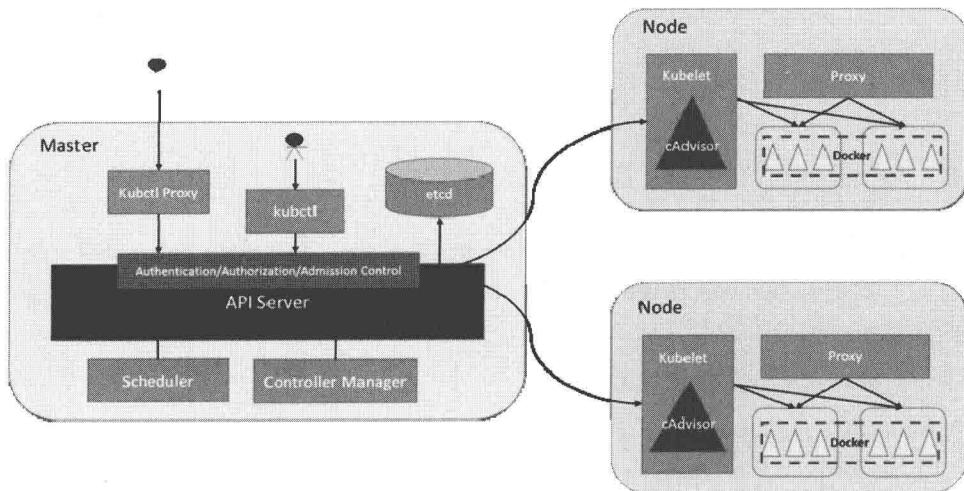


图 1.14 Kubernetes 的系统架构图

etcd 是高可用的 key/value 存储系统，用于持久化存储集群中所有的资源对象，例如集群中的 Node、Service、Pod、RC、Namespace 等。API Server 则提供了操作 etcd 的封装接口 API，

以 REST 方式提供服务，这些 API 基本上都是集群中资源对象的增删改查及监听资源变化的接口，比如创建 Pod、创建 RC，监听 Pod 的变化等接口。API Server 是连接其他所有服务组件的枢纽。下面我们以 RC 与相关 Service 创建的完整流程为例，来说明 Kubernetes 里各个服务（组件）的作用以及它们之间的交互关系。

我们通过 Kubectl 提交一个创建 RC 的请求（假设 Pod 副本数为 1），该请求通过 API Server 被写入 etcd 中，此时 Controller Manager 通过 API Server 的监听资源变化的接口监听到这个 RC 事件，分析之后，发现当前集群中还没有它所对应的 Pod 实例，于是根据 RC 里的 Pod 模板定义生成一个 Pod 对象，通过 API Server 写入 etcd 中，接下来，此事件被 Scheduler 发现，它立即执行一个复杂的调度流程，为这个新 Pod 选定一个落户的 Node，可称这个过程为绑定（Pod Binding），然后又通过 API Server 将这一结果写入到 etcd 中，随后，目标 Node 上运行的 Kubelet 进程通过 API Server 监测到这个“新生的”Pod 并且按照它的定义，启动该 Pod 并任劳任怨地负责它的下半生，直到 Pod 的生命走到尽头。

随后，我们通过 Kubectl 提交一个映射到该 Pod 的 Service 的创建请求，Controller Manager 会通过 Label 标签查询到相关联的 Pod 实例，然后生成 Service 的 Endpoints 信息并通过 API Server 写入到 etcd 中。接下来，所有 Node 上运行的 Proxy 进程通过 API Server 查询并监听 Service 对象与其对应的 Endpoints 信息，建立一个软件方式的负载均衡器来实现 Service 访问到后端 Pod 的流量转发功能。

从上面的分析来看，Kubernetes 的各个组件的功能是很清晰的。

- ◎ **API Server:** 提供了资源对象的唯一操作入口，其他所有组件都必须通过它提供的 API 来操作资源数据，通过对相关的资源数据“全量查询”+“变化监听”，这些组件可以很“实时”地完成相关的业务功能，比如某个新的 Pod 一旦被提交到 API Server 中，Controller Manager 就会立即发现并开始调度。
- ◎ **Controller Manager:** 集群内部的管理控制中心，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，比如根据 RC 的定义完成 Pod 的复制或移除，以确保 Pod 实例数符合 RC 副本的定义；根据 Service 与 Pod 的管理关系，完成服务的 Endpoints 对象的创建和更新；其他诸如 Node 的发现、管理和状态监控、死亡容器所占磁盘空间及本地缓存的镜像文件的清理等工作也是由 Controller Manager 完成的。
- ◎ **Scheduler:** 集群中的调度器，负责 Pod 在集群节点中的调度分配。
- ◎ **Kubelet:** 负责本 Node 节点上的 Pod 的创建、修改、监控、删除等全生命周期管理，同时 Kubelet 定时“上报”本 Node 的状态信息到 API Server 里。
- ◎ **Proxy:** 实现了 Service 的代理及软件模式的负载均衡器。

客户端通过 Kubectl 命令行工具或 Kubectl Proxy 来访问 Kubernetes 系统，在 Kubernetes 集群内部的客户端可以直接使用 Kubectl 命令管理集群。Kubectl Proxy 是 API Server 的一个反向代理，在 Kubernetes 集群外部的客户端可以通过 Kubectl Proxy 来访问 API Server。

API Server 内部有一套完备的安全机制，包括认证、授权及准入控制等相关模块。API Server 在收到一个 REST 请求后，会首先执行认证、授权和准入控制的相关逻辑，过滤掉非法请求，然后将请求发送给 API Server 中的 REST 服务模块去执行资源的具体操作逻辑。

在 Node 节点运行的 Kubelet 服务中内嵌了一个 cAdvisor 服务，cAdvisor 是谷歌的另外一个开源项目，用于实时监控 Docker 上运行的容器的性能指标，在第 4 章会详细介绍它。

1.6 Kubernetes 安装与配置

1.6.1 安装 Kubernetes

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

安装 Kubernetes 对软件和硬件的系统要求如表 1.1 所示。

表 1.1 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 1GB 内存 Node: 至少 1 core 和 1GB 内存	Master: 2 core 和 2GB 内存 Node: 由于要运行 Docker，所以应根据需要的容器数量进行配置
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
Docker	1.3 版本及以上 下载和安装说明： https://www.docker.com	1.8 版本
etcd	2.0 版本及以上 下载和安装说明： https://github.com/coreos/etcd/releases	2.2 版本

最简单的安装方法是从 Kubernetes 官网下载编译好的二进制包，如图 1.15 所示，本书基于 Kubernetes 1.0 版本进行说明。下载地址为：<https://github.com/GoogleCloudPlatform/kubernetes/releases>。

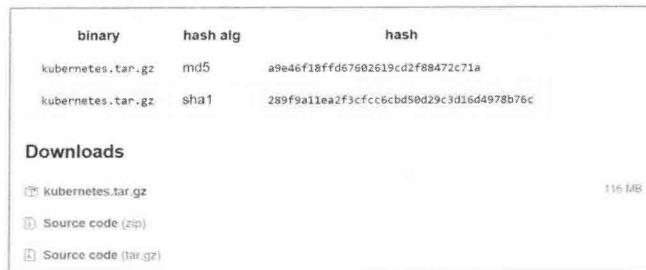


图 1.15 GitHub 上 Kubernetes 的下载页面

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

解压缩后，`server` 子目录中的 `kubernetes-server-linux-amd64.tar.gz` 文件包含了全部 Kubernetes 需要运行的服务程序文件。文件列表如表 1.2 所示。

表 1.2 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>	命令行控制程序
<code>kubelet</code>	Kubelet 主程序
<code>kube-proxy</code>	proxy 主程序
<code>kube-scheduler</code>	scheduler 主程序
<code>kube-scheduler.docker_tag</code>	scheduler docker 镜像的 tag
<code>kube-scheduler.tar</code>	scheduler docker 镜像文件

Kubernetes Master 节点安装部署 `kube-apiserver`、`kube-controller-manager`、`kube-scheduler` 等服务进程。我们可以使用 `Kubectl` 作为客户端与 Master 进行交互操作。在工作 Node 上仅需使用 `Kubelet` 和 `kube-proxy`。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。

1.6.2 配置和启动 Kubernetes 服务

Kubernetes 并没有过多地依赖软件，使用二进制文件加上必要的启动参数直接运行即可完

成 Kubernetes 服务的启动。为了便于管理，常见的做法是将 Kubernetes 服务程序配置为 Linux 的系统服务。

本节以 Red Hat 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，Red Hat Linux 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，在内网系统中建议关闭防火墙服务：

```
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，则需要将系统服务文件中相应的文件路径修改正确即可），然后进行对服务的配置。

在下面对服务启动参数的说明中主要介绍了必要的参数，每个服务的启动参数还有很多，可以在第 4 章中找到完整的说明，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 kube-apiserver、kube-controller-manager、kube-scheduler 服务

1) kube-apiserver 服务

在 /usr/lib/systemd/system 目录下创建 kube-apiserver.service 文件。

假设 etcd 服务已经安装并正确启动，则 kube-apiserver 将依赖于 etcd 服务，需要在该文件中加入 After=etcd.service 和 Wants=etcd.service。

接下来，将 kube-apiserver 的启动参数放在 kube-apiserver.service 文件中，并通过使用配置文件中定义的环境变量来指定各参数。

```
# cd /usr/lib/systemd/system/  
#  
# more kube-apiserver.service  
[Unit]  
Description=Kubernetes API Server  
Documentation=https://github.com/GoogleCloudPlatform/kubernetes  
After=etcd.service  
Wants=etcd.service  
  
[Service]  
EnvironmentFile=-/etc/kubernetes/config  
EnvironmentFile=-/etc/kubernetes/apiserver  
User=kube  
ExecStart=/usr/bin/kube-apiserver \  
    $KUBE_LOGTOSTDERR \  
    $KUBE_LOG_LEVEL \  
    --v=2
```

```
$KUBE_etcd_SERVERS \
$KUBE_API_ADDRESS \
$KUBE_API_PORT \
$KUBELET_PORT \
$KUBE_ALLOW_PRIV \
$KUBE_SERVICE_ADDRESSES \
$KUBE_ADMISSION_CONTROL \
$KUBE_API_ARGS
Restart=on-failure
Type=notify
LimitNOFILE=65536

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

配置文件存放于/etc/kubernetes 目录中，其中 config 文件的内容为所有服务都需要的参数。与 kube-apiserver 相关的参数放置于 apiserver 文件中。

配置文件 config 的内容包括：log 设置、是否允许运行具有特权模式的 Docker 容器及 Master 所在的地址等。

```
# cd /etc/kubernetes

# more config
### 
# kubernetes system config
#
# The following values are used to configure various aspects of all
# kubernetes services, including
#
#   kube-apiserver.service
#   kube-controller-manager.service
#   kube-scheduler.service
#   kubelet.service
#   kube-proxy.service
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR= "--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL= "--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV= "--allow_privileged=false"

# How the controller-manager, scheduler, and proxy find the apiserver
KUBE_MASTER= "--master=http://kubernetes-master:8080"
```

配置文件 apiserver 的内容包括：绑定主机的 IP 地址、端口号、etcd 服务地址、Service 所

需的 Cluster IP 池、一系列 admission 控制策略等。

```
# cat apiserver
###
# kubernetes system config
#
# The following values are used to configure the kube-apiserver
#
# The address on the local server to listen to.
KUBE_API_ADDRESS= "--insecure-bind-address=0.0.0.0"

# The port on the local server to listen on.
KUBE_API_PORT= "--insecure-port=8080"

# Comma separated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS= "--etcd_servers=http://127.0.0.1:4001"

# Address range to use for services
KUBE_SERVICE_ADDRESSES= "--service-cluster-ip-range=10.254.0.0/16"

# default admission control policies
KUBE_ADMISSION_CONTROL=
--admission_control=NamespaceAutoProvision,LimitRanger,SecurityContextDeny

# Add your own!
KUBE_API_ARGS= ""
```

2) kube-controller-manager 服务

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

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

[Service]
EnvironmentFile=/etc/kubernetes/config
EnvironmentFile=/etc/kubernetes/controller-manager
ExecStart=/usr/bin/kube-controller-manager \
    $KUBE_LOGTOSTDERR \
    $KUBE_LOG_LEVEL \
```

```
$KUBE_MASTER \
$KUBE_CONTROLLER_MANAGER_ARGS
Restart=on-failure
LimitNOFILE=65536

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

同样，可以将配置文件 controller-manager 放置于/etc/kubernetes 目录中。通常无须特别的参数设置。

3) kube-scheduler 服务

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

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

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

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

配置文件/etc/kubernetes/scheduler 通常无须特别的参数设置。

配置完成后，通过 systemctl start 命令启动这 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>` 来验证服务启动的状态，“running” 表示启动成功。

到此，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。

1) Kubelet 服务

与 Master 服务的配置相同，在`/usr/lib/systemd/system` 目录创建 `kubelet.service` 文件对 Kubelet 服务进行配置，它依赖于 Docker 服务。

```
[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/config  
EnvironmentFile=-/etc/kubernetes/kubelet  
ExecStart=/usr/bin/kubelet \  
    $KUBE_LOGTOSTDERR \  
    $KUBE_LOG_LEVEL \  
    $KUBELET_API_SERVER \  
    $KUBELET_ADDRESS \  
    $KUBELET_PORT \  
    $KUBELET_HOSTNAME \  
    $KUBE_ALLOW_PRIV \  
    $KUBELET_ARGS  
Restart=on-failure  
  
[Install]  
WantedBy=multi-user.target
```

配置文件`/etc/kubernetes/kubelet` 的内容包括：绑定主机 IP 地址、端口号、apiserver 的地址及其他参数。

```
###  
# kubernetes kubelet (minion) config  
  
# The address for the info server to serve on (set to 0.0.0.0 or " " for all interfaces)  
KUBELET_ADDRESS=" --address=0.0.0.0 "
```

```
# The port for the info server to serve on
KUBELET_PORT=" --port=10250 "

# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME=" --hostname_override=node1 "

# location of the api-server
KUBELET_API_SERVER=" --api_servers=http://kubernetes-master:8080 "

# Add your own!
KUBELET_ARGS=" "
```

2) kube-proxy 服务

创建 kube-proxy.service 文件，该服务依赖于 Linux 的 network 服务。

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

[Service]
EnvironmentFile=/etc/kubernetes/config
EnvironmentFile=/etc/kubernetes/proxy
ExecStart=/usr/bin/kube-proxy \
    $KUBE_LOGTOSTDERR \
    $KUBE_LOG_LEVEL \
    $KUBE_MASTER \
    $KUBE_PROXY_ARGS
Restart=on-failure
LimitNOFILE=65536

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

配置文件/etc/kubernetes/proxy 无须特别的参数设置。

Kubelet 和 kube-proxy 都需要的配置文件 config 的内容示例如下。

```
# cd /etc/kubernetes

# more config
###
# kubernetes system config
#
# The following values are used to configure various aspects of all
# kubernetes services, including
#
```

```
# kube-apiserver.service
# kube-controller-manager.service
# kube-scheduler.service
# kubelet.service
# kube-proxy.service
# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privileged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"

# How the controller-manager, scheduler, and proxy find the apiserver
KUBE_MASTER="--master=http://kubernetes-master:8080"
```

配置完成后，通过 `systemctl` 启动服务：

```
# 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 自注册的机制，在 Master 上查看各 Node 的状态（`$ kubectl get nodes`），状态为 Ready 表示 Node 向 Master 注册成功。等所有 Node 的状态都为 Ready 之后，一个 Kubernetes 集群就启动完成了。接下来就可以使用配置文件创建 RC、Pod、Service 等对象来部署 Docker 容器应用集群了。

1.6.3 Kubernetes 的版本升级

Kubernetes 的升级很简单，按如下步骤执行即可完成。

- ① 通过官网下载最新版本的二进制包 `kubernetes.tar.gz`，解压缩。
- ② 停止 Master 和 Nodes 上的 Kubernetes 相关服务。
- ③ 将新版可执行文件复制到 Kubernetes 安装的目录下，覆盖旧版文件。
- ④ 重启各 Kubernetes 服务。

1.6.4 内网中的 Kubernetes 相关配置

Kubernetes 在能够访问 Internet 网络的环境中使用非常方便，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，用户通过租用一定空间来部署 Kubernetes 集群也很容易。

但是，许多企业内部由于安全性原因无法访问 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 的 Registry 工具可以方便地创建一个 Private Registry。

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` 的镜像来完成对 Pod 网络的配置。

该镜像存在于谷歌网站 <http://gcr.io> 中，可以通过一台能够连上 Internet 的服务器将其下载，导出文件，再保存到私有 Docker Registry 中去。

之后，需要给每台 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：

```
###  
# kubernetes kubelet (minion) config  
  
# The address for the info server to serve on (set to 0.0.0.0 or " " for all interfaces)  
KUBELET_ADDRESS="--address=0.0.0.0"  
  
# The port for the info server to serve on  
KUBELET_PORT="--port=10250"  
  
# You may leave this blank to use the actual hostname  
KUBELET_HOSTNAME="--hostname_override=node1"  
  
# location of the api-server  
KUBELET_API_SERVER="--api_servers=http://kubernetes-master:8080"  
  
# Add your own!  
KUBELET_ARGS="--pod_infra_container_image=docker.intranet.com:5000/google_containers/pause:latest"
```

修改 Kubelet 配置文件后，重启 Kubelet 服务：

```
sudo systemctl restart kubelet
```

通过以上设置就完成了在无法访问 Internet 的内网环境中搭建了一个企业内部的私有云平台。

1.6.5 Kubernetes 对 Docker 镜像的要求——启动命令前台执行

在使用 Docker 时，通常使用 docker run 命令创建并启动一个容器。在 Kubernetes 系统中对容器的要求是：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 UNIX 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。

以本章留言板例子中 redis-master 镜像的启动命令为例：

```
redis-server /etc/Redis/Redis.conf
```

该命令表示 redis-server 进程将一直在前台执行。

另外，guestbook-php-frontend 镜像的默认启动命令为：

```
apache2-foreground
```

apache2-foreground 同样是一个前台执行命令。

许多传统的应用程序都被设计为服务的形式在后台执行，例如 UNIX 系统中大量使用的以 nohup 方式运行的程序。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> 中文档的说明。

第2章

Kubernetes 核心原理

本章通过 5 节来讲述 Kubernetes 核心原理，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 组件的运行机制。最后，深入分析安全机制和网络原理。

2.1 Kubernetes API Server 分析

总结下来，Kubernetes API Server 有如下功能和地位：

- (1) 提供了集群管理的 API 接口；
- (2) 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；
- (3) 拥有完备的集群安全机制。

在本节主要针对上述第（1）、（2）点进行分析，在 2.4 节会对第（3）点进行详细分析。

2.1.1 如何访问 Kubernetes API

Kubernetes API 通过一个叫作 Kubernetes apiserver 的进程提供服务，这个进程运行在单个 kubernetes-master 节点上。在默认情况下，该进程包含如下两个端口。

- 1) 本地端口
 - (1) 该端口用于接收 HTTP 请求；
 - (2) 该端口的默认值为 8080，可以通过修改 API Server 的启动参数 “--insecure-port” 的值

来修改该默认值；

(3) 默认的 IP 地址是“localhost”，通过修改 API Server 的启动参数“`--insecure-bind-address`”的值来修改该 IP 地址；

(4) 非认证或授权（Authentication or Authorization）的 HTTP 请求通过该端口访问 API Server。

2) 安全端口（Secure Port）

(1) 该端口的默认值为 6443，通过修改 API Server 的启动参数“`--secure-port`”的值可以修改该默认值；

(2) 默认的 IP 地址为非本地（Non-Localhost）网络接口，通过 API Server 的启动参数“`--bind-address`”设置该值；

(3) 该端口用于接收 HTTPS 请求；

(4) 用于基于 Token 文件或客户端证书及 HTTP Base 的认证；

(5) 用于基于策略的授权；

(6) Kubernetes 默认不启动 HTTPS 安全访问机制。

我们既可以通过编程方式访问 API Server，也可以通过 curl 命令直接访问它。假如 API Server 的地址是\$APISERVER（格式为：ip:port），则用下面的命令即可调用它的“Versions” REST 接口：

```
$ curl $APISERVER/api --header "Authorization: Bearer $TOKEN" -insecure \
{ \
  "versions": [ \
    "v1" \
  ] \
}
```

参数\$TOKEN 为用户的 Token，用于安全验证机制。此外，Kubernetes 还提供了一个代理程序——Kubectl Proxy，它既能作为 Kuberntes API Server 的反向代理，也能作为普通客户端访问 API Server 的代理。假如通过 Master 节点的 8080 端口来启动该代理程序，则可以运行下面的命令：

```
kubectl proxy --port=8080 &
```

验证代理是否正常工作，可以通过访问该代理的“Versions” REST 接口进行测试：

```
curl http://localhost:8080/api/ \
{
  "versions": [ \
    "v1" \
  ] \
}
```

作为 API Server 的反向代理，可以通过它开发或限制对外暴露的功能。作为客户端访问 API

Server的普通代理，认证部分完全可以交由它去处理。

Kubernetes及各开源社区为开发人员提供了各种语言版本的Client Libraries，通过这些Client Libraries或HTTP REST程序包，开发人员能够使用自己编写的程序访问Kubernetes API Server。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。

此外，Kubernetes还提供了命令行工具Kubectl，用它来将API Server的API包装成简单的命令集供我们使用。Kubectl的实现原理很简单，它首先把用户的输入转换为对API Server的REST API调用（包含API地址和访问参数），然后发起远程调用，并将调用结果输出。因此，我们可以认为Kubectl是API Server的一个客户端工具。通过全面学习和掌握Kubectl的用法，我们基本上可以弄明白API Server的大部分API接口的参数、意义及作用。

Kubectl命令如下：

```
kubectl [command] [options]
```

command列表如表2.1所示。

表2.1 command列表

命 令	说 明
get	显示一个或多个资源的信息
describe	详细描述某个资源的信息
create	通过文件名或标准输入创建一个资源
update	通过文件名或标准输入修改一个资源
delete	通过文件名、标准输入、资源的ID或标签删除资源
namespace	设置或查看当前请求的命名空间
logs	打印在Pod中的容器的日志信息
rolling-update	对一个给定的ReplicationController执行滚动更新（Rolling Update）
scale	调节Replication Controller副本数量
exec	在某个容器内执行某条命令
port-forward	为某个Pod设置一个或多个端口转发
proxy	运行Kubernetes API Server代理
run	在集群中运行一个独立的镜像（Image）
stop	通过ID或资源名称删除一个资源
expose	将资源对象暴露为Kubernetes Service
label	修改某个资源上的标签（Label）
config	修改集群的配置信息
cluster-info	显示集群信息
api-versions	显示API版本信息
version	打印Kubectl和API Server版本信息
help	帮助命令

option 列表如表 2.2 所示。

表 2.2 option 列表

参 数	说 明
--alsologtostderr=false	记录日志到标准错误输出及文件
--api-version= " "	用于告知 API Server 的 Kubectl 使用的 API 版本信息
--certificate-authority= " "	证书文件的访问路径
--client-certificate= " "	客户端证书文件路径（包括目录和文件名）
--client-key= " "	客户端私钥文件路径（包括目录和文件名）
--cluster= " "	指定集群的名称
--context= " "	Kubectl 配置文件上下文的名字
-h, --help=false	是否支持 Kubectl 帮助命令
--insecure-skip-tls-verify=false	如果该值为 true，则将不校验服务端证书，这将使 HTTPS 连接不安全
--kubeconfig= " "	Kubectl 配置文件的访问路径
--log-backtrace-at=:0	当日志内容达到 N 行时，产生一个“stack trace”
--log-dir=	指定日志文件的目录
--log-flush-frequency=5s	两个 log flush 操作之间最大的时间间隔，单位为秒。
--logtostderr=true	打印日志到标准错误输出，代替写入文件
--match-server-version=false	要求服务端版本和客户端版本匹配
--namespace= " "	指定命名空间
--password= " "	使用基本认证访问 API Server 时用到的密码
-s, --server= " "	指定 Kubernetes API Server 的地址
--stderrthreshold=2	设置将日志写入标准错误输出的阈值
--token= " "	使用 Token 方式访问 API Server 时用到的令牌
--user= " "	访问 API Server 的用户名
--username= " "	使用基本认证访问 API Server 时用到的用户名
--v=0	V logs 的日志级别
--validate=false	如果该值为 true，则在发送一个请求前使用一个“schema”校验输入信息
--vmodule=	用于设置过滤日志的模式，各模式之间用逗号隔开

2.1.2 通过 API Server 访问 Node、Pod 和 Service

图 2.1 列出了访问集群 API Server 及集群内资源对象互访的情况。

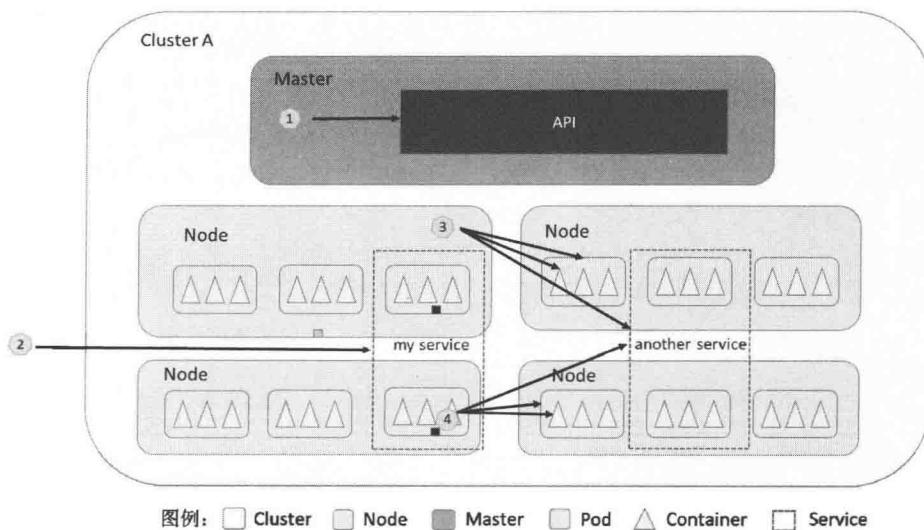


图 2.1 访问 Kubernetes 集群

以下是对图 2.1 中各种访问途径的详细说明（每个数字表示一种访问的途径）。

(1) 集群内部各组件、应用或集群外部应用访问 API Server。

(2) 集群外部系统访问 Service。

(3) 该种情况包含:

- ① 集群内跨节点访问 Pod;
- ② 集群内跨节点访问容器;
- ③ 集群内跨节点访问 Service。

(4) 该种情况包含:

- ④ 集群内的容器访问 Pod;
- ⑤ 集群内的容器访问其他集群内的容器;
- ⑥ 集群内的容器访问 Service。

本节中只涉及通过 API Server 访问 Node、Pod 和 Service 的情况，对其他情况请参考其他章节。

集群外系统可以通过访问 API Server 提供接口管理 Node 节点，该接口的路径为 /api/v1/proxy/nodes/{name}，其中 {name} 为节点的名称或 IP 地址。由于该接口是 REST 接口，因此支持增删改查方法。API Server 除提供上述接口外，还提供如下接口：

```
/api/v1/proxy/nodes/{name}/pods      #列出节点内所有 Pod 信息  
/api/v1/proxy/nodes/{name}/stats     #列出节点内物理资源的统计信息  
/api/v1/proxy/nodes/{name}/spec      #列出节点概要信息
```

前面所列的三个接口，只有在该节点的 Kubelet 启动时包含--enable-server=true 参数时，才能被访问。如果 Node 的 Kubelet 进程在启动时包含--enable-debugging-handlers=true 参数，那么 API Server 会包含如下访问接口：

```
/api/v1/proxy/nodes/{name}/run    #在节点上运行某个容器，参考 Docker 的 run 命令  
/api/v1/proxy/nodes/{name}/exec   #在节点上的某个容器中运行某条命令，参考 Docker 的 exec 命令  
/api/v1/proxy/nodes/{name}/attach  #在节点上 attach 某个容器，参考 Docker 的 attach 命令  
/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/、rhsm/、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，具体使用情况参考 godoc 的说明
```

通过 API Server 不仅可以管理 Pod，而且可以通过 API Server 访问 Pod 提供的服务，访问的接口列表如下：

```
/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  
  
/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
```

如果某个 Service 包含 kubernetes.io/cluster-service: " true " 和 kubernetes.io/name: "\$CLUSTERSERVICENAME" 标签，其中\$CLUSTERSERVICENAME 为集群 Service 的名称，那么用户可以通过 API Server 的/api/v1/proxy/namespaces/{namespace}/services/{name} 接口访问该 Service。也可以通过 API Server 的 /api/v1/proxy/namespaces/{namespace}/services/{name}/{path:*} 接口访问 Service 后端的 Pod 中的容器提供的服务。例如：

```
apiVersion: v1  
kind: Service  
metadata:  
  name: mywebservice  
  namespace: kube-system  
  labels:  
    kubernetes.io/cluster-service: " true "  
    kubernetes.io/name: " myclusterwebapp "  
spec:  
  selector:  
    k8s-smp: mywebpod
```

```

clusterIP: 10.2.0.100
ports:
- name: dns-tcp
  port: 53
protocol: TCP

```

该例子创建了一个名称为“myclusterwebapp”的集群 Service，用户通过 API Server 的 /api/v1/proxy/namespaces/kub-system/services/myclusterwebapp 接口能够管理该 Service。假如该 Service 后端的容器在 Web 应用的根路径下包含一个名为 helloWorld 的 Servlet，则用户可以通过 /api/v1/proxy/namespaces/kub-system/services/myclusterwebapp/helloWorld 访问该 Servlet。

2.1.3 集群功能模块之间的通信

从图 2.2 中可以看出，API Server 作为集群的核心，负责集群各功能模块之间的通信。集群内的功能模块通过 API Server 将信息存入 etcd，其他模块通过 API Server（用 get、list 或 watch 方式）读取这些信息，从而实现模块之间的信息交互。比如，Node 节点上的 Kubelet 每隔一个时间周期，通过 API Server 报告自身状态，API Server 收到这些信息后，将节点状态信息保存到 etcd 中。Controller Manager 中的 Node Controller 通过 API Server 定期读取这些节点状态信息，并做相应处理。又比如，Scheduler 监听到某个 Pod 创建的信息后，检索所有符合该 Pod 要求的节点列表，并将 Pod 绑定到节点列表中最符合要求的节点上；如果 Scheduler 监听到某个 Pod 被删除，则调用 API Server 删除该 Pod 资源对象。Kubelet 监听 Pod 信息，如果监听到 Pod 对象被删除，则删除本节点上的相应的 Pod 实例；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 实例等。

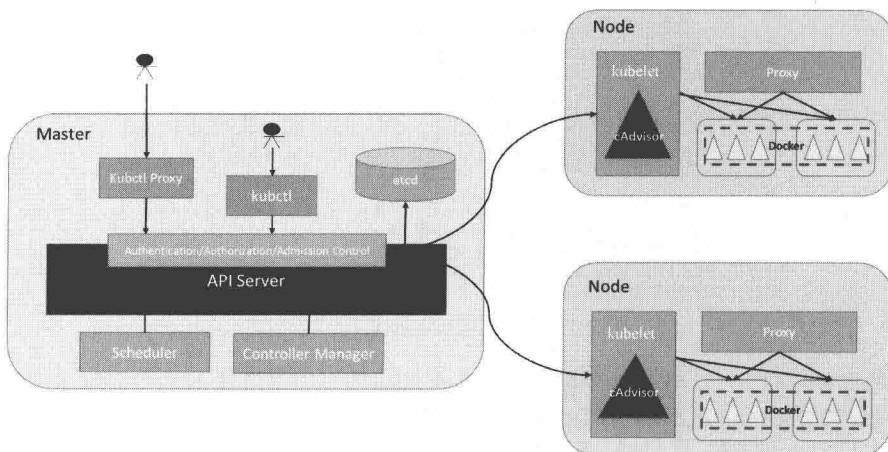


图 2.2 Kubernetes 结构图

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 获取指定资源对象信息（通过 list 及 watch 方式），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。

2.2 调度控制原理

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）等的管理并执行自动化修复流程，确保集群处于预期的工作状态。比如在出现某个 Node 意外宕机时，Controller Manager 会在集群的其他节点上自动补齐 Pod 副本。

如图 2.3 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 等多个控制器，Controller Manager 是这些控制器的核心管理者。一般来说，智能系统和自动系统通常会通过一个操纵系统来不断修正系统的状态。在 Kubernetes 集群中，每个 Controller 就是一个操纵系统，它通过 API Server 监控系统的共享状态，并尝试着将系统状态从“现有状态”修正到“期望状态”。本章的前面几个小节介绍了 Controller Manager 的这些 Controller 的原理。

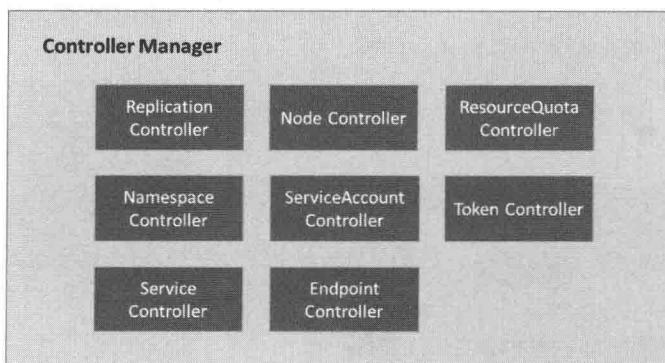


图 2.3 Controller Manager 结构图

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绑定到某个合适的 Node 上。本章最后会介绍 Kubernetes Scheduler 调度器的基本原理。

2.2.1 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”，以便于后续讨论。

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 都保持一定数量的 Pod 副本处于正常运行状态。如果该类 Pod 的 Pod 副本数量太多，则 Replication Controller 会销毁一些 Pod 副本；反之 Replication Controller 会添加 Pod 副本，直到该类 Pod 的 Pod 副本数量达到预设的副本数量。最好不要越过 RC 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，这样就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。

Service 可能由被不同 RC 管理的多个 Pod 副本组成，在 Service 的整个生命周期里，由于需要发布不同版本的 Pod，因此希望不断有旧的 RC 被销毁，新的 RC 被创建。Service 自身及它的客户端应该不需要关注 RC。

Replication Controller 管理的对象是 Pod，因此其操作和 Pod 的状态及重启策略息息相关。Pod 的状态值列表如表 2.3 所示。

表 2.3 Pod 的状态值列表

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
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或正在启动或重启
succeeded	Pod 内所有容器均成功中止，且不会再重启
failed	Pod 内所有容器均已退出，且至少有一个容器因为发生错误而退出

Pod 的重启策略包含：Always、OnFailure 和 Never。当 Pod 的重启策略 `RestartPolicy = Always` 时，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，用户或 Replication Controller 会销毁 Pod 对象。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。当 Pod 副本变成 failed 状态或被删除，且其 `RestartPolicy = Always` 时，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。

为了理解 Replication Controller 的机制，我们需要先进一步理解 RC。被 RC 管控的所有

Pod 实例都是通过 RC 里定义的 Pod 模板（Templet）创建的，该模板包含 Pod 的标签属性，同时 RC 里包含一个标签选择器（Label Selector），Selector 的值表明了该 RC 所关联的 Pod。RC 会保证每个由它创建的 Pod 都包含与它的标签选择器相匹配的 label。通过这种标签选择器技术，Kubernetes 实现了一种简单地过滤、选择资源对象的机制，并且这个机制被 Kubernetes 大量使用。另外，通过 RC 创建的 Pod 副本在初始阶段状态是一致的，从某种意义上讲是可以完全互相替换的。这种特性非常适合副本无状态服务，当然，RC 同样可以用于构建有状态的服务。

下面我们来创建一个 RC 来加深对上述原理的理解，RC 的定义如下所示：

```
{
  "kind": "ReplicationController", "apiVersion": "v1",
  "metadata": { "name": "tecip1src", "labels": { "name": "tecip1src" } },
  "spec": {
    "replicas": 3, "selector": { "name": "tecip1s" },
    "template": {
      "metadata": { "labels": { "name": "tecip1s" } },
      "spec": {
        "containers": [
          {
            "name": "ecipt1000", "image": "10.248.12.110:1180/ecipt",
            "ports": [ { "containerPort": 1100, "hostPort": 1100 } ],
            "env": [
              { "name": "INSTID", "value": "1000" },
              { "name": "INSTPORT", "value": "1100" },
              { "name": "DBUSER", "value": "sn100" },
              { "name": "DBPSW", "value": "sn100" },
              { "name": "TWSECU_RMIPort", "value": "2000" },
              { "name": "TWSECU_SERVER", "value": "10.248.12.108" }
            ]
          }
        ]
      }
    }
  }
}
```

这个 RC 创建了一个包含了一个容器的 Pod——tecip1s，该 Pod 包含 3 个副本。

关于 Pod 模板的问题，我们可以这样来理解：模板就像一个模具，模具制作出来的东西一旦离开模具，它们之间就再也没关系。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。

理解了 RC 的作用，我们就容易理解 Replication Controller 了，它的职责有：

- (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 属性值来调整 Pod 的副本数量。

Kubernetes 的各个模块职责明确且简单有效。

副本控制器的常用使用模式如下。

(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。例如，通过下列命令可以实现手动修改名为 foo 的 RC 的副本数为 3：

```
kubectl scale --replicas=3 replicationcontrollers foo
```

(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。下面是手动调用滚动更新的例子代码：

```
kubectl rolling-update frontend-v1 -f frontend-v2.json
```

在上面对应用滚动更新的讨论中，我们发现一个应用在滚动更新时，可能存在多个版本的 Release。事实上，在生产环境中一个已经发布的应用程序存在多个 Release 版本是很正常的现象。通过 RC 的标签选择器，我们能很方便地实现对一个应用的多版本 Release 进行追踪。假设一个 Kubernetes 服务对象 (Service) 包含多个 Pod，这些 Pod 的 labels 均为 tier=frontend, environment=prod，Pod 的数量为 10 个，如果你希望拿出其中一个 Pod 用于测试新功能，则你可以这样做：

(1) 首先，创建一个 RC，并设置其 Pod 副本数量为 9，其标签选择器设置为 tier=frontend, environment=prod, track=stable；

(2) 然后，通过滚动更新来创建一个 RC，并设置其 Pod 副本数量为 1 (就是那个用于测试的 Pod)，其标签选择器设置为 tier=frontend, environment=prod, track=canary。这样，Service 就

同时覆盖了测试版和稳定版的 Pod，实现了对应用程序多版本 Release 的追踪。

2.2.2 Node Controller

Node Controller 负责发现、管理和监控集群中的各个 Node 节点。Kubelet 在启动时通过 API Server 注册节点信息，并定时向 API Server 发送节点信息。API Server 接收到这些信息后，将这些信息写入 etcd。存入 etcd 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。下面列举节点信息的内容：

```
{  
    "kind": "Node",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "e2e-test-wojtekt-minion-etd6",  
        "selfLink": "/api/v1/nodes/e2e-test-wojtekt-minion-etd6",  
        "uid": "a7e89222-e8e5-11e4-8fde-42010af09327",  
        "resourceVersion": "379",  
        "creationTimestamp": "2015-04-22T11:49:39Z"  
    },  
    "spec": {  
        "externalID": "15488322946290398375"  
    },  
    "status": {  
        "capacity": {  
            "cpu": "1",  
            "memory": "1745152Ki"  
        },  
        "conditions": [  
            {  
                "type": "Ready",  
                "status": "True",  
                "lastHeartbeatTime": "2015-04-22T11:58:17Z",  
                "lastTransitionTime": "2015-04-22T11:49:52Z",  
                "reason": "kubelet is posting ready status"  
            }  
        ],  
        "addresses": [  
            {  
                "type": "ExternalIP",  
                "address": "104.197.49.213"  
            },  
            {  
                "type": "LegacyHostIP",  
                "address": "104.197.49.213"  
            }  
        ]  
    }  
}
```

```
        "address" : "104.197.20.11"
    }
],
"nodeInfo": {
    "machineID": "",
    "systemUUID": "D59FA3FA-7B5B-7287-5E1A-1D79F13CB577",
    "bootID": "44a832f3-8cfb-4de5-b7d2-d66030b6cd95",
    "kernelVersion": "3.16.0-0.bpo.4-amd64",
    "osImage": "Debian GNU/Linux 7 (wheezy)",
    "containerRuntimeVersion": "docker://1.5.0",
    "kubeletVersion": "v0.15.0-484-g0c8ee980d705a3-dirty",
    "kubeProxyVersion": "v0.15.0-484-g0c8ee980d705a3-dirty"
}
}
```

如图 2.4 所示，Node Controller 通过 API Server 定期读取这些信息，然后做如下处理。

(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、第一次收到节点 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。

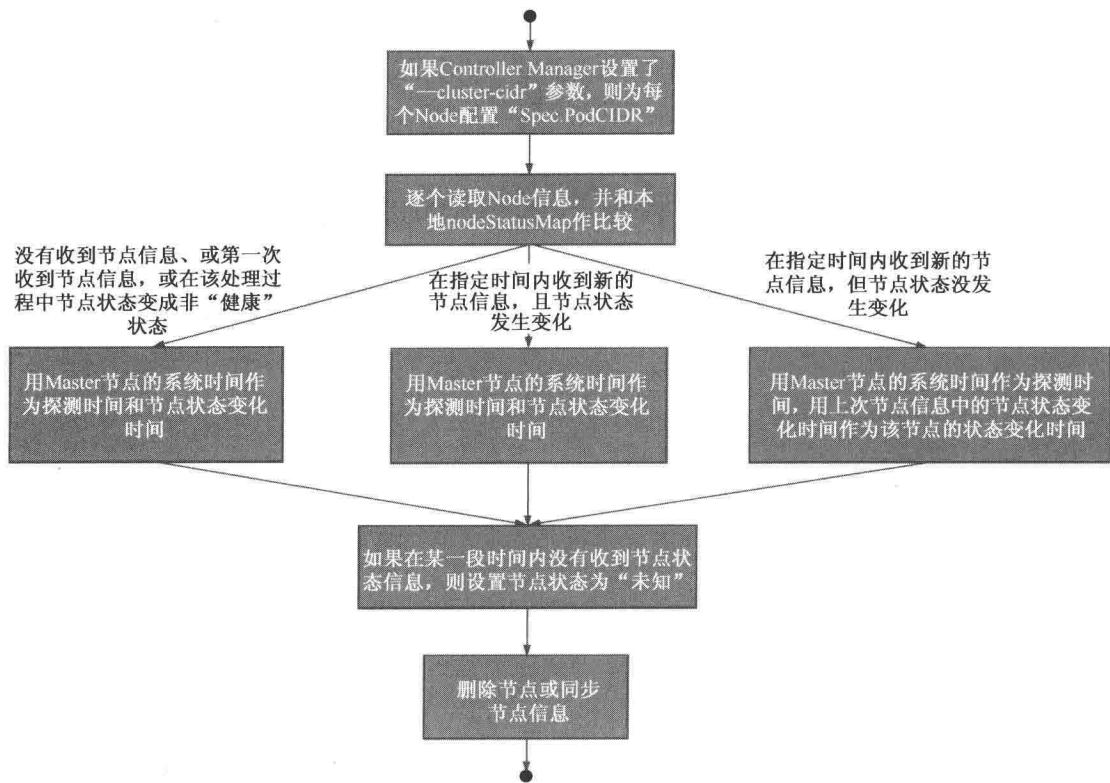


图 2.4 Node Controller 流程图

2.2.3 ResourceQuota Controller

作为容器集群的管理平台，Kubernetes 也提供了资源配额管理（ResourceQuota Controller）这一高级功能，资源配额管理确保了指定的对象在任何时候都不会超量占用系统资源，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额管理。

- (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（可以用于多租户）级别的资源限制，包括：
 - ① Pod 数量；

- ① Replication Controller 数量;
- ② Service 数量;
- ③ ResourceQuota 数量;
- ④ Secret 数量;
- ⑤ 可持有的 PV (Persistent Volume) 数量。

Kubernetes 的配额管理是通过准入机制 (Admission Control) 来实现的, 与配额相关的两种准入控制器是 LimitRanger 与 ResourceQuota, 其中 LimitRanger 作用于 Pod 和 Container 上, ResourceQuota 则作用于 Namespace 上。此外, 如果定义了资源配置, 则 kube-scheduler 在 Pod 调度过程中也会考虑这一因素, 确保 Pod 调度不会超出配额限制。

ResourceQuota Controller 负责实现 Kubernetes 的资源配置管理, 如图 2.5 所示。用户通过 API Server 为 Namespace 维护 ResourceQuota 对象, API Server 将该对象保存到 etcd 中。所有 Pod、Service、RC、Secret 和 Persistent Volume 资源对象的实时状态通过 API Server 保存到 etcd 中, ResourceQuota Controller 在计算资源使用总量时会用到这些信息。

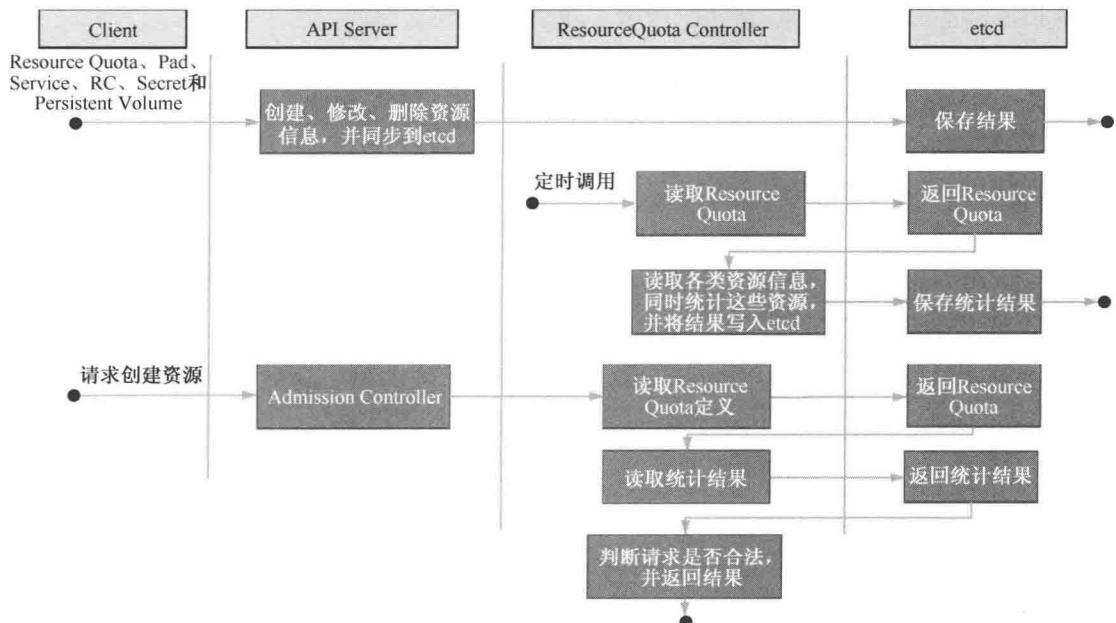


图 2.5 ResourceQuota Controller 流程图

ResourceQuota Controller 以 Namespace 作为分组统计单元, 通过 API Server 定时读取 etcd 中每个 Namespace 里定义的 ResourceQuota 信息, 计算 Pod、Service、RC、Secret 和 Persistent

Volume 等资源对象的总数，以及所有 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）。

用户通过 API Server 请求创建或修改资源时，API Server 会调用 Admission Controller 的 ResourceQuota 插件，该插件会读取前面写入 etcd 的配额统计结果，如果某项资源的配额已经被使用完，则此请求会被拒绝。

2.2.4 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（设置删除期限，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 Adminission 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（即 DeletionTimestam 属性被设置），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。

2.2.5 ServiceAccount Controller 与 Token Controller

ServiceAccount Controller 与 Token Contoller 是与安全相关的两个控制器。Service Account Controller 在 Controller manager 启动时被创建。它监听 Service Account 的删除事件和 Namespace 的创建、修改事件。如果在该 Service Account 的 Namespace 中没有 default Service Account，那么 Service Account Controller 为该 Service Account 的 Namespace 创建一个 default Service Account。

我们在 API Server 的启动参数中添加“--admission_control=ServiceAccount”后，API Server 在启动时会自己创建一个 key 和 crt（见/var/run/kubernetes/apiserver.crt 和 apiserver.key），然后在启动./kube-controller-manager 时添加参数 service_account_private_key_file=/var/run/kubernetes/

`apiserver.key`, 这样启动 Kubernetes Master 后, 我们就会发现在创建 Service Account 时系统会自动为其创建一个 Secret。

如果 Controller manager 在启动时指定参数为 `service-account-private-key-file`, 而且该参数所指定的文件包含一个 PEM-encoded 编码的 RSA 算法的私钥, 那么, Controller manager 会创建 Token Controller 对象 (线程)。

Token Controller 对象监听 Service Account 的创建、修改和删除事件, 并根据事件的不同做不同的处理。如果监听到的事件是创建和修改 Service Account 事件, 则读取该 Service Account 的信息; 如果该 Service Account 没有 Service Account Secret (即用于访问 API Server 的 Secret), 则用前面提及的私钥为该 Service Account 创建一个 JWT Token, 将该 Token 和 ROOT CA (如果启动时参数指定了该 ROOT CA) 放入新建的 Secret 中, 将该新建的 Secret 放入该 Service Account 中, 同时修改 etcd 中 Service Account 的内容。如果监听到的事件是删除 Service Account 事件, 则删除与该 Service Account 相关的 Secret。

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。如果监听到的事件是创建和修改 Secret 事件, 那么读取该 Secret 中 annotation 所指定的 Service Account 信息, 并根据需要为该 Secret 创建一个和其 Service Account 相关的 Token; 如果监听到的事件是删除 Secret 事件, 则删除 Secret 和相关的 Service Account 的引用关系。

2.2.6 Service Controller 与 Endpoint Controller

在学习 Service Controller 之前, 让我们先深入了解一下 Kubernetes Service, 它是一个定义 Pod 集合的抽象, 或者被访问者看作一个访问策略, 有时也被称作微服务。

Kubernetes 中的 Service 是一种资源对象, 和 Pod 相似。同所有其他资源对象一样, 可以通过 API Server 的 POST 接口创建一个新的实例。在下面的例子代码中创建了一个名为 “my-service” 的 Service, 它包含一个标签选择器, 通过该标签选择器选择所有包含标签为 “app=MyApp”的 Pod 作为该 Service 的 Pod 集合。Pod 集合中的每个 Pod 的 80 端口被映射到节点本地的 9376 端口, 同时 Kubernetes 指派一个集群 IP (即前面提到的虚拟 IP) 给该 Service。内容列表如下:

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "my-service"
  },
  "spec": {
```

```

    "selector": {
        "app": "MyApp"
    },
    "ports": [
        {
            "protocol": "TCP",
            "port": 80,
            "targetPort": 9376
        }
    ]
}
}
}

```

图 2.6 列出了 Service 是如何访问到后端的 Pod 的。在创建 Service 时，如果指定了标签选择器（在 spec.selector 域中指定），那么系统会自动创建一个和该 Service 同名的 Endpoint 资源对象。该 Endpoint 资源对象包含一个地址（Addresses）和端口（Ports）集合。这些 IP 地址和端口号即通过标签选择器过滤出来的 Pod 的访问端点。Kubernetes 支持通过 TCP 和 UDP 去访问这些 Pod 的地址，默认使用 TCP。

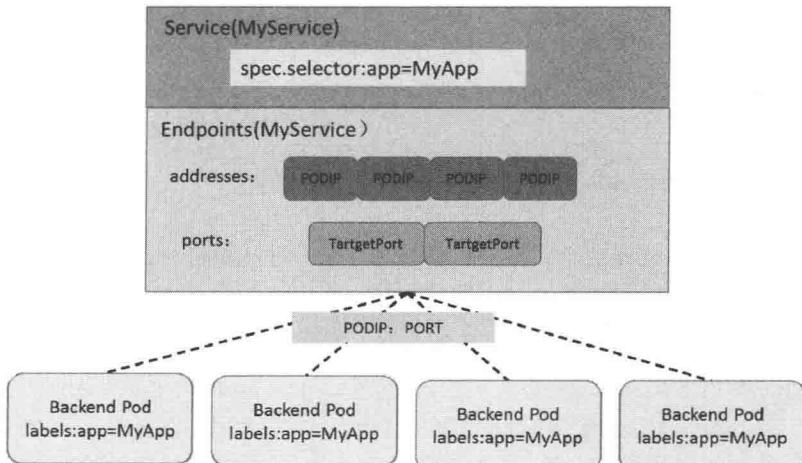


图 2.6 Service、Endpoint、Pod 的关系

在某些特殊场景下，例如将一个外部数据库作为 Service 的后端，或将在另外一个集群或 Namespace 中的服务作为服务的后端，需要创建一个不带标签选择器的 Service，如下所示：

```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "my-service"
    },
}
```

```

    "spec": {
      "ports": [
        {
          "protocol": "TCP",
          "port": 80, #service的port
          "targetPort": 9376 #POD中某个容器的某个端口号
        }
      ]
    }
}

```

由于该例子创建的是一个不带标签选择器的 Service，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。Endpoint 创建的文件内容如下：

```

{
  "kind": "Endpoints",
  "apiVersion": "v1",
  "metadata": {
    "name": "my-service"
  },
  "subsets": [
    {
      "addresses": [
        { "IP": "1.2.3.4" }
      ],
      "ports": [
        { "port": 80 }
      ]
    }
  ]
}

```

如图 2.7 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。

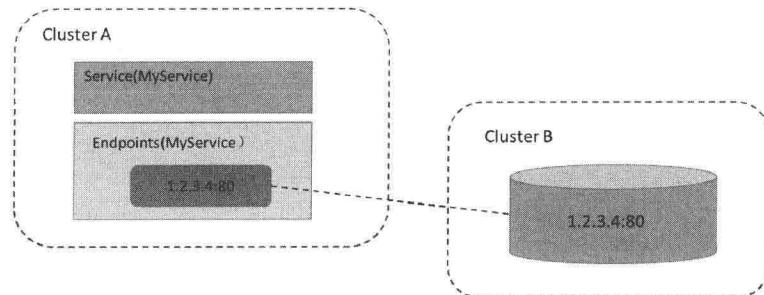


图 2.7 不带标签选择器的 Service

如何通过虚拟 IP 访问到后端 Pod 呢？

在 Kubernetes 集群中的每个节点上都运行着一个叫作“kube-proxy”的进程，该进程会观察 Kubernetes Master 节点添加和删除“Service”和“Endpoint”的行为，如图 2.8 第①步所示。kube-proxy 为每个 Service 在本地主机上开一个端口（随机选择）。任何访问该端口的连接都被代理到相应的一个后端 Pod 上。kube-proxy 根据 Round Robin 算法及 Service 的 Session 粘连（SessionAffinity）决定哪个后端 Pod 被选中，如图 2.8 第②步所示。最后，如图 2.8 第③步所示，kube-proxy 在本机的 Iptables 中安装相应的规则，这些规则使得 Iptables 将捕获的流量重定向到前面提及的随机端口。通过该端口流量再被 kube-proxy 转到相应的后端 Pod 上。

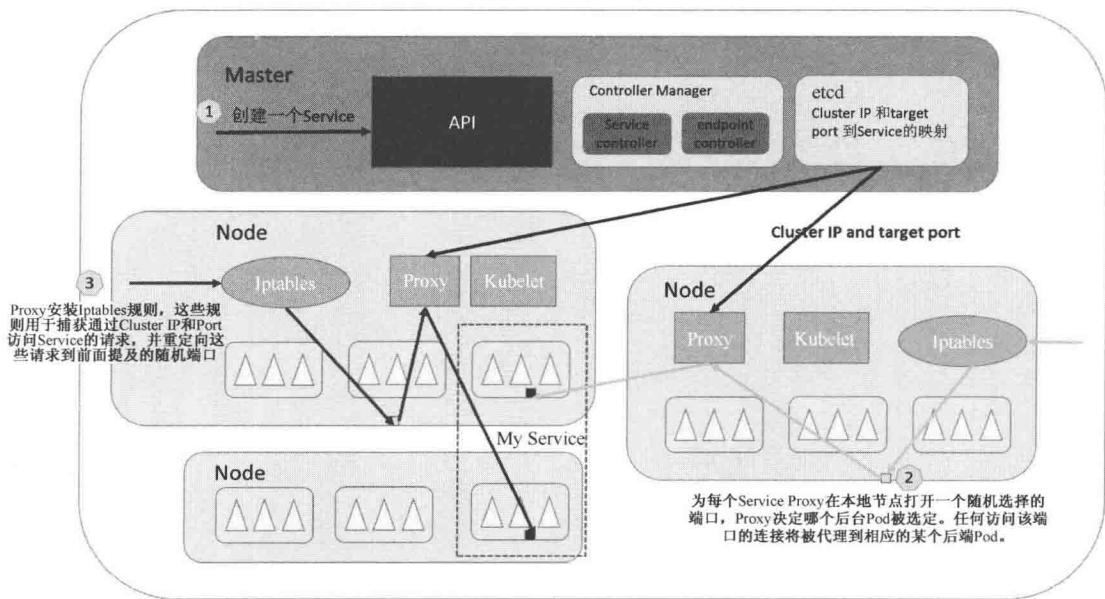


图 2.8 创建和访问 Service

如上所述，在创建了服务后，服务 Endpoint 模型会创建后端 Pod 的 IP 和端口列表（包含在 Endpoints 对象中），kube-proxy 就是从这个 Endpoint 列表中选择服务后端的。集群内的节点通过虚拟 IP 和端口能够访问 Service 后端的 Pod。

在默认情况下，Kubernetes 会为 Service 指定一个集群 IP（或虚拟 IP、cluster IP），但在某些情况下，用户希望能够自己指定该集群 IP。为了给 Service 指定集群 IP，用户只需要在定义 Service 时，在 Service 的 spec.clusterIP 域中设置所需要的 IP 地址即可。为 Service 指定的 IP 地址必须在集群的 CIDR 范围内。如果该 IP 地址是非法的，那么 API Server 会返回 422 HTTP 状态码，表明 IP 地址值非法。

Kubernetes 支持两种主要的模式来找到 Service，一个是容器的 Service 环境变量，另一个是 DNS。

在创建一个 Pod 时，Kubelet 在该 Pod 中的所有容器中为当前所有 Service 添加一系列环境变量。Kubernetes 既支持 Docker links 变量，也支持形如“`{SVCNAME}_SERVICE_HOST`”和“`{SVCNAME}_SERVICE_PORT`”的变量。其中“`{SVCNAME}`”是大写的 Service Name，同时 Service Name 包含的“-”符号会转化成“_”符号。例如，名称为“redis-master”的 Service，它对外暴露 6379 TCP 端口，且集群 IP 地址为 10.0.0.11。Kubelet 会为新建的容器添加如下环境变量：

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

通过环境变量来找到 Service 会带来一个不好的结果，即任何被某个 Pod 所访问的 Service，必须先于该 Pod 被创建。否则和这个后创建的 Service 相关的环境变量，将不会被加入该 Pod 的容器中。

另一个通过名字找到服务的方式是 DNS。DNS 服务器通过 Kubernetes API 监控与 Service 相关的活动。当监控到添加 Service 时，DNS 服务器为每个 Service 创建一系列 DNS 记录。例如，在 Kubernetes 集群的“my-ns”Namespace 中有一个叫作“my-service”的 Service，用“my-service”通过 DNS 应该能够访问到“my-ns” Namespace 中的后端 Pod。如果在其他 Namespace 中访问这个 Service，则用“my-service.my-ns”来查找该 Service。DNS 返回的查找结果是集群 IP（虚拟 IP、cluster IP）。

Kubernetes 也支持 DNS SRV (Service) 被命名端口的记录。如果“my-service.my-ns” Service 有一个名为“http”的端口，则可以用“_http._tcp.my-service.my-ns”通过 DNS 服务器找到对应的 Pod 暴露的端口。

集群外部用户希望 Service 能够提供一个供集群外部用户访问的 IP 地址，甚至是公网 IP 地址，通过该 IP 来访问集群内的 Service。Kubernetes 通过两种方式来实现上述需求，一个是“NodePort”，另一个是“LoadBalancer”。每个 Service 定义的“spec.type”域作为定义 Service 类型的地方，该域包括如下 3 个合法值。

- ① ClusterIP：默认值，仅使用集群内部虚拟 IP（集群 IP、Cluster IP）。
- ② NodePort：使用虚拟 IP（集群 IP、Cluster IP），同时通过在每个节点上暴露相同的端口来暴露 Service。

- ⑤ LoadBalancer: 使用虚拟 IP (集群 IP、Cluster IP) 和 NodePort, 同时请求云服务商作为转向 Service 的负载均衡器。

注意, 在 Kubernetes 1.0 中, NodePort 既支持 TCP 也支持 UDP, 而 LoadBalancer 仅能支持 TCP。

如果在定义 Service 时, 设置 spec.type 的值为 “NodePort”, 则 Kubernetes Master 节点将为 Service 的 NodePort 指派一个端口范围 (默认为 30000~32767), 每个节点的 kube-proxy 将重定向请求到 Service 的端口 (在 spec.ports.targetPort 中指定的端口)。如果你希望为 NodePort 指定一个端口, 则可以在 Service 定义中通过指定 spec.ports.nodePort 域的值实现, 你所指定的端口必须在前面提及的端口范围内。如图 2.9 中的①、②、③和④所示。

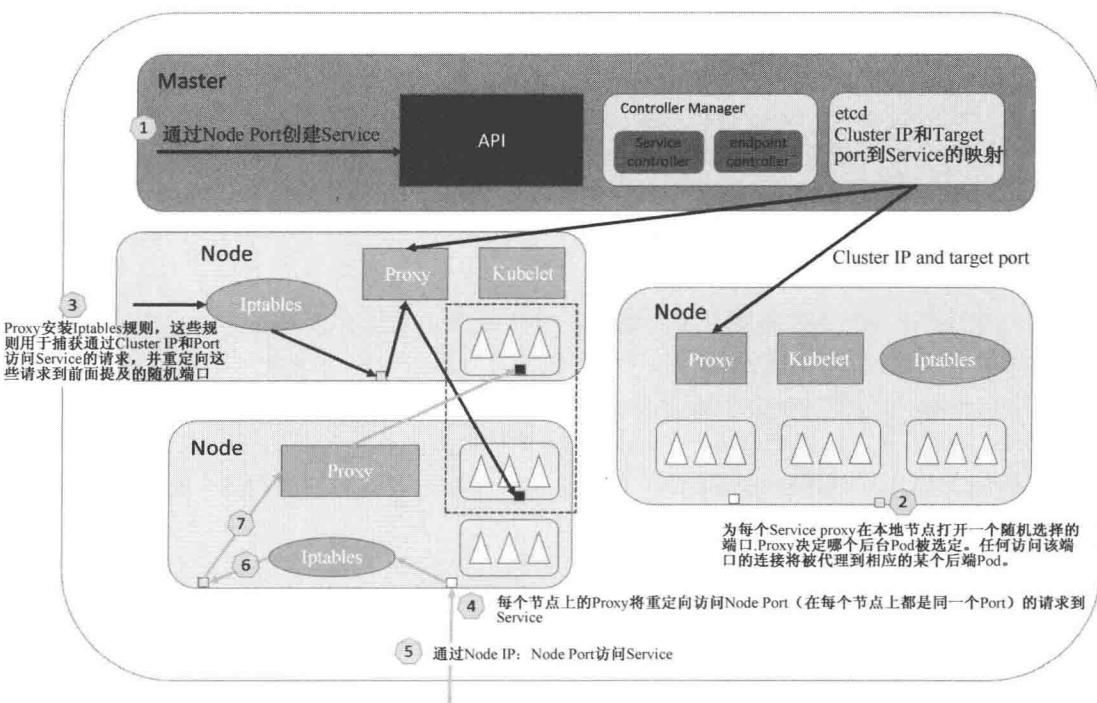


图 2.9 创建和访问 Node Port Service

通过 NodePort 类型的 Service, 集群外的用户可以通过任意节点的 IP 及 spec.ports.nodePort 域指定的端口号访问 Service 后端的 Pod, 如图 2.9 中的⑤、⑥和⑦所示。这使得开发者能够自由地设置其负载均衡器, 自由地配置没有完全被 Kubernetes 支持的云环境, 甚至仅仅需要对外暴露一个或多个节点 IP。

Service Controller 监控 Service 的变化，如果发生变化的 Service 是 LoadBalancer 类型的（`externalLoadBalancers=true`），则 Service Controller 确保外部的 LoadBalancer 被相应地创建和删除。Service Controller 定期检查集群的 Service，确保相应的外部 LoadBalancer 存在。Service Controller 定期检查集群节点，确保外部 LoadBalancer 被更新。

Endpoints Controller 通过 Store 来缓存 Service 和 Pod 信息，它监控 Service 和 Pod 的变化。Endpoints Controller 通过 API Server 监控 etcd 的“/registry/services”目录（用 Watch 和 List 方式）。如果监测到 Service 被删除，则删除和该 Service 同名的 Endpoint 对象；根据该 Service 信息获得相关的 Pod 列表，根据 Service 和 Pod 对象列表创建一个新的 Endpoint 的 subsets 对象。如果判断出是新建或修改 Service，那么用 Service 的 name 和 labels 及上面创建的 subsets 对象创建一个 Endpoint 对象，并同步到 etcd。

Endpoints Controller 通过 API Server 监控 etcd 的“/registry/pods”目录（以 Watch 和 List 方式）。如果检测到添加或删除 Pod，则从本地缓存中找到与该 Pod 相关的 Service 列表，为该 Service 列表中的 Service 逐个创建 Endpoint 并同步到 etcd。如果监测到 Pod 被修改，则从本地缓存中找到和新 Pod 相关的 Service 列表，以及和旧 Pod 相关的 Service 列表。合并这两个 Service 列表，逐个为合并后的 Service 列表中的 Service 创建 Endpoints 对象并同步到 etcd 中。

2.2.7 Kubernetes Scheduler

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 服务进程接管后续工作，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 2.10 所示。

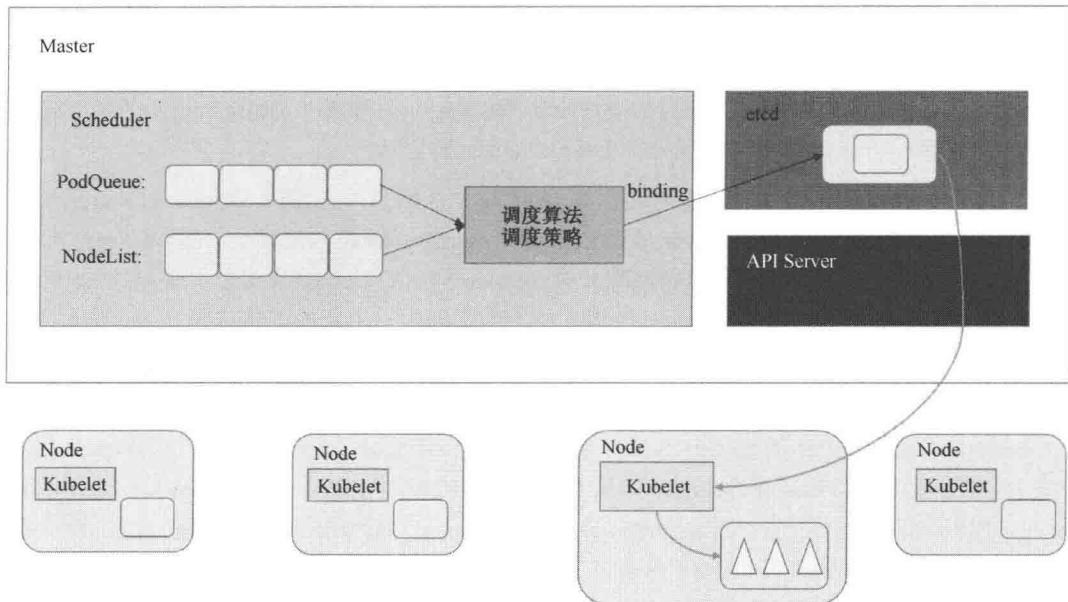


图 2.10 Scheduler 流程

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，在第一步的基础上，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)
```

2.3 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 cAdvise 监控容器和节点资源。

2.3.1 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。如果该参数值为 true，那么 Kubelet 将试着通过 API Server 注册自己。作为自注册，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。

2.3.2 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 端点的数据时间间隔，默认为 20 秒。

(3) API Server：Kubelet 通过 API Server 监听 etcd 目录，同步 Pod 清单。

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：

- ① 为容器计算一个 hash 值，然后用容器的名字去 Docker 查询对应容器的 hash 值。若查找到容器，且两者 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；
- ② 如果容器被中止了，且容器没有指定的 restartPolicy (重启策略)，则不做任何处理；
- ③ 调用 Docker Client 下载容器镜像，调用 Docker Client 运行容器。

2.3.3 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。

2.3.4 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的使用情况。

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 2.11 是 cAdvisor 的一个截图。



图 2.11 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。

2.4 安全机制的原理

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 6 个方面来说明集群的安全机制。

2.4.1 Authentication 认证

Kubernetes 对 API 调用使用 CA (Client Authentication)、Token 和 HTTP Base 方式实现用户认证。

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 2.12 所示，SSL 双向认证大概包含以下几个步骤。

(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 通信双方的客户器端向 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，并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 2.12 CA 认证流程

如上所述的是双向认证 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密的（这并不影响 SSL 过程的安全性）密码方案。

通过上述内容可知，使用 CA 认证的应用需包含一个 CA 认证机构（外部或企业自身）。通过该机构给服务器端下发根证书、服务端证书和私钥文件，给客户端下发根证书、客户端证书和私钥文件。因此 API Server 的三个参数“`--client-ca-file`”“`--tls-cert-file`”和“`--tls-private-key-file`”分别指向根证书文件、服务端证书文件和私钥文件。API Server 客户端应用的三个启动参数（例如 Kubectl 的三个参数“`certificate-authority`”“`client-certificate`”和“`client-key`”），或客户端应用的 kubeconfig 配置文件中的配置项“`certificate-authority`”“`client-certificate`”和“`client-key`”，分别指向根证书文件、客户端证书文件和私钥文件。

Kubernetes 的 CA 认证方式通过添加 API Server 的启动参数“`--client_ca_file=SOMEFILE`”实现，其中“`SOMEFILE`”为认证授权文件，该文件包含一个或多个证书颁发机构(CA Certificates Authorities)。

Token 认证方式通过添加 API Server 的启动参数“`--token_auth_file=SOMEFILE`”实现，其中“`SOMEFILE`”指的是存储 Token 的 Token 文件。目前，Token 认证中 Token 是永久有效的，而且 Token 列表不能被修改，除非重启 API Server。Kubernetes 计划在未来的版本中 Token 认证的 Token 将为短期有效，按需要生成 Token，而不是像现在这样存储在文件中。Token 文件格式为一个包含三列的 CSV 格式文件，该文件的第一列为 Token，第二列为用户名，第三列为用户 UID。当使用 Token 认证方式从 HTTP 客户端访问 API Server 时，HTTP 请求头中的 Authorization 域必须包含“`Bearer SOMETOKEN`”的值，其中“`SOMETOKEN`”为该访问客户端持有的 Token。例如 Token 文件中的内容为：

```
1kjqweroiuou,Thomas,8x7d1kklzseertyywx
```

用 CURL 去访问该 API Server：

```
curl $APISERVER/api --header "Authorization: Bearer 1kjqweroiuou" --insecure
{
    "versions": [
        "v1"
    ]
}
```

基本认证方式是通过添加 API Server 的启动参数“`--basic_auth_file=SOMEFILE`”实现的，其中“`SOMEFILE`”指的是用于存储用户和密码信息的基本认证文件。当前，基本认证文件中的用户和密码信息永远有效，同时密码不能改变，除非重新启动 API Server。基本认证文件格式为一个包含三列的 CSV 格式文件，该文件的第一列为密码，第二列为用户名，第三列为用户 UID。当使用基本认证方式从 HTTP 客户端访问 API Server 时，HTTP 请求头中的 Authorization

域必须包含“Basic BASE64ENCODEDUSER:PASSWORD”的值，其中“BASE64ENCODEDUSER:PASSWORD”为该访问客户 base64 加密算法加密后的用户名和密码。比如用户 Thomas 的密码为 Thomas，通过下面的代码访问 API Server：

```
tmp=`base64 "Thomas:Thomas"`
curl $APISERVER/api --header "Authorization: Basic $tmp" --insecure
{
  "versions": [
    "v1"
  ]
}
```

2.4.2 Authorization 授权

在 Kubernetes 中，授权（Authorization）是认证（Authentication）后的一个独立步骤，作用于 API Server 主要端口的所有 HTTP 访问。授权流程不作用于只读端口，在计划中只读端口在不久之后将被删除。授权流程通过访问策略比较请求上下文的属性（例如用户名、资源和 Namespace）。在通过 API 访问资源之前，必须通过访问策略进行校验。访问策略通过 API Server 的启动参数“--authorization_mode”配置，该参数包含如下三个值：

```
--authorization_mode=AlwaysDeny
--authorization_mode=AlwaysAllow
--authorization_mode=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；“ABAC”表示使用用户配置的授权策略去管理访问 API Server 的请求，ABAC（Attribute-Based Access Control）为基于属性的访问控制。

在 Kubernetes 中，一个 HTTP 请求包含如下 4 个能被授权进程识别的属性。

- ➊ 用户名（代表一个已经被认证的用户的字符型用户名）；
- ➋ 是否是只读请求（REST 的 GET 操作是只读的）；
- ➌ 被访问的是哪一类资源，例如访问 Pod 资源/api/v1/namespaces/default/pods；
- ➍ 被访问对象所属的 Namespace，如果这被访问的资源不支持 Namespace，则是空字符串。

由于通过过多的属性来实现访问控制会增加管理的复杂度，因此 Kubernetes 仅用四个属性来实现访问控制，并不希望添加更多的属性去实现访问控制。API Server 接收到请求后，会读取该请求中所带的前面提及的四个属性。如果该请求中不带某些属性，则这些属性的值将根据值的类型设置成零值（例如为字符串类型属性设置一个空字符串；为布尔型属性设置为 false；

为数值类型属性设置 0)。

如果选用 ABAC 模式,那么需要通过设置 API Server 的“--authorization_policy_file=SOME_FILENAME”参数来指定授权策略文件,其中“SOME_FILENAME”为授权策略文件。授权策略文件的每一行都是一个 JSON 对象,该 JSON 对象是一个 Map,这个 Map 内不包含 List 和 Map。每行都是一个“策略对象”。策略对象包含下面 4 个属性:

- ◎ 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。

没被设置的属性,将被等同于根据值的类型设置成零值(例如为字符串类型属性设置一个空字符串;为布尔型属性设置 false;为数值类型属性设置 0)。

授权策略文件中的策略对象的一个未设置属性,表示匹配 HTTP 请求中该属性的任何值。对请求的 4 个属性值和授权策略文件中的所有策略对象逐个匹配,如果至少有一个策略对象被匹配上,则该请求将被授权通过。

例如。

- (1) 允许用户 alice 做任何事情: { "user": "alice" }。
- (2) 用户 Kubelet 指定读取资源 Pods: { "user": "kubelet", "resource": "pods", "readonly": true }。
- (3) 用户 Kubelet 能读和写资源 events: { "user": "kubelet", "resource": "events" }。
- (4) 用户 bob 只能读取 Namespace "myNamespace" 中的资源 Pods: { "user": "bob", "resource": "pods", "readonly": true, "ns": "myNamespace" }。

该例子的授权策略文件 ad.json 的内容如下:

```
{ "user": "alice" }
{ "user": "kubelet", "resource": "pods", "readonly": true}
{ "user": "kubelet", "resource": "events" }
{ "user": "bob", "resource": "pods", "readonly": true, "ns": "myNamespace" }
```

2.4.3 Admission Control 准入控制

Admission Control 是用于拦截所有经过认证和鉴权后的访问 API Server 请求的可插入代码

(或插件)。这些可插入代码运行于 API Server 进程中，在被调用前必须被编译成二进制文件。在请求被 API Server 接收前，每个 Admission Control 插件按配置顺序执行。如果其中的任意一个插件拒绝该请求，就意味着这个请求被 API Server 拒绝，同时 API Server 反馈一个错误信息给请求发起方。

在某些情况下，Admission Control 插件会使用系统配置的默认值去改变进入集群对象的内容。此外，Admission Control 插件可能会改变请求处理所使用的资源的配额，比如增加请求处理的资源配额。

通过配置 API Server 的启动参数“admission_control”，在该参数中加入需要的 Admission Control 插件列表，各插件的名称之间用逗号隔开。例如：

```
--admission_control=NamespaceAutoProvision,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota
```

Admission Control 的插件列表如表 2.4 所示。

表 2.4 Admission Control 的插件列表

名 称	说 明
AlwaysAdmit	允许所有请求通过
AlwaysDeny	拒绝所有请求，一般用于测试
DenyExecOnPrivileged	拦截所有带有 SecurityContext 属性的 Pod 的请求，拒绝在一个特权容器中执行命令
ServiceAccount	配合 Service Account Controller 使用，为设定了 Service Account 的 Pod 自动管理 Secret，使得 Pod 能够使用相应的 Secret 下载 Image 和访问 API Server
SecurityContextDeny	不允许带有 SecurityContext 属性的 Pod 存在，SecurityContext 属性用于创建特权容器
ResourceQuota	在 Namespace 中做资源配置限制
LimitRanger	限制 Namespace 中的 Pod 和 Container 的 CPU 和内存配额
NamespaceExists	读取请求中的 Namespace 属性，如果该 Namespace 不存在，则拒绝该请求
NamespaceAutoProvision (deprecated)	读取请求中的 Namespace 属性，如果该 Namespace 不存在，则尝试创建该 Namespace
NamespaceLifecycle	该插件限制访问处于中止状态的 Namespace，禁止在该 Namespace 中创建新的内容。当 NamespaceLifecycle 和 NamespaceExists 能够合并成一个插件后，NamespaceAutoProvision 就会变成 deprecated

在上述列表中列出了所有的 Adminission Control 插件，大部分比较易于理解，接下来着重介绍 SecurityContextDeny、ResourceQuota 及 LimitRanger 这三个插件。

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，禁止通过 API Server 管理配置了下列两项配置的 Pod：

```
spec.containers.securityContext.seLinuxOptions
spec.containers.securityContext.runAsUser
```

2) ResourceQuota

Kubernetes 的 ResourceQuota 插件不仅能够限制某个 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。Kubernetes 通过两类方式实现资源配置限制，一个是资源对象个数的配额限制，另一个是资源使用总量的配额限制。在 API Server 的启动参数中加入“--admission_control=ResourceQuota”后，该插件生效。该插件和 ResourceQuota 对象一起实现了资源配置管理。

如果在某个 Namespace 中包含 ResourceQuota 对象，那么该对象会在该 Namespace 中生效。资源对象个数配额限制是指某个 Namespace 中资源对象的最大数量限制。表 2.5 列出了所有的资源对象数量配额限制。

表 2.5 所有的资源对象数量配额限制

名 称	说 明
pods	最大 Pod 数量
services	最大 Services 数量
replicationcontrollers	最大 RC 数量
resourcequotas	最大 ResourceQuota 数量
Secrets	最大 Secret 数量
persistentvolumeclaims	最大 PersistentVolume 申明数量

例如表 2.5 中列出的 Pod 资源对象数量的配额限制，限制了某个 Namespace 中所创建的 Pod 的最大数量。

资源使用总量配额限制是指某个 Namespace 中资源使用量的最大限制。表 2.6 列出了所有的资源对象总量配额限制。

表 2.6 所有的资源对象总量配额限制

名 称	说 明
cpu	所有容器 CPU 使用最大总量
memory	所有容器内存使用最大总量

例如，表 2.6 中列出的 CPU 资源使用总量配额限制，限制了某个 Namespace 所有 Pod 中容器 resources.limits.cpu 域值的总和的最大值。

下面的代码表示在 myspace Namespace 中创建一个 ResourceQuota 对象：

```
$ cat <<EOF > quota.json
{
  "apiVersion": "v1",
  "kind": "ResourceQuota",
```

```

"metadata": {
    "name": "quota"
},
"spec": {
    "hard": {
        "memory": "1Gi",
        "cpu": "20",
        "pods": "10",
        "services": "5",
        "replicationcontrollers": "20",
        "resourcequotas": "1"
    }
}
}
EOF
$ kubectl create -f quota.json namespace=myspace

```

3) LimitRanger

Kubernetes 的 LimitRanger 插件用于列举 Namespace 中各类资源的最小限制、最大限制及默认值，它针对 Namespace 资源的每个个体限制每个个体的资源配额。其限制的资源类型包括 Pod 和 Container 两类。

表 2.7 列出了资源类型为 Container 的资源限制。

表 2.7 Container 的资源限制

资源名称	说明
cpu	每个容器 CPU 的最大/最小值
memory	每个容器内存的最大/最小值

表 2.8 列出了资源类型为 Pod 的资源限制。

表 2.8 Pod 的资源限制

资源名称	说明
cpu	每个 Pod CPU 的最大/最小值
memory	每个 Pod 内存的最大/最小值

如果为某资源指定了默认值，那么它可能作用于即将创建的资源。例如：如果没有设置容器的资源请求的默认值，但通过设置 LimitRange 对象中 Container CPU 的默认值，那么这个 LimitRange 中的默认值将会作用于即将创建的容器。

如果为某资源指定了最小值，那么它可能作用于即将创建的资源。例如：如果没有设置容器的资源请求的最小值，但通过设置 LimitRange 对象中 Container CPU 的最小值，那么这个 LimitRange 中的最小值将会作用于即将创建的容器。

在 API Server 的启动参数中加入“`--admission_control=LimitRanger`”后，该插件生效。该插件和 LimitRange 对象一起实现资源限制管理。下面的例子在 myspace 中创建一个 LimitRange 对象，如下所示：

```
$ cat <<EOF > limits.yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: mylimits
spec:
  limits:
  - max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 250m
      memory: 6Mi
    type: Pod
  - default:
      cpu: 250m
      memory: 100Mi
    max:
      cpu: "2"
      memory: 1Gi
    min:
      cpu: 250m
      memory: 6Mi
    type: Container
$ kubectl create -f limits.yaml - namespace=myspace
```

2.4.4 Secret 私密凭据

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。

Kubernetes 在 Pod 创建时，如果该 Pod 指定了 Service Account，那么为该 Pod 自动添加包含凭证信息的 Secrets，用于访问 API Server 和下载 Image。该功能可以通过 Admission Control 添加或失效，然而如果需要以安全的方式去访问 API Server，则建议开启该功能。

下面的例子用于创建一个 Secret：

```
$ kubectl namespace myspace
$ cat <<EOF > 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) 在创建 Pod 时，指定 Pod 的 spec. ImagePullSecrets 来引用它。

第 1 种使用方式在下一节中将会做详细说明。下面的例子展示第 2 种使用方式，展示如何将一个 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 2.13 所示。

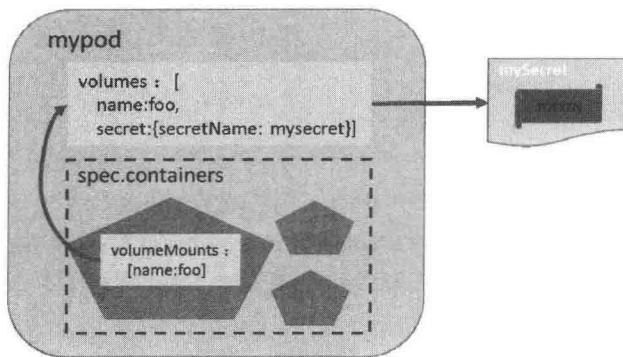


图 2.13 挂载 Secret 到 Pod

第三种使用方式是手动使用 `imagePullSecret`, 其流程如下:

(1) 执行 `login` 命令, 登录私有 Registry:

```
$ docker login localhost:1180
```

输入用户名和密码, 如果是第一次登录系统, 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:

```
$ cat > image-pull-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
data:
  .dockercfg: eyAiaHR0cHM6Ly9pbmRleC5kb2NrZXIuaW8vdjEvIjogeyAiYXV0aCI6ICJab
UZyWlhCaGMzTjNiM0prTVRJSyIsICJlbWFpbCI6ICJqZG91QGV4YW1wbGUuY29tIiB9IH0K
  type: kubernetes.io/dockercfg
EOF
$ kubectl create -f image-pull-secret.yaml
```

(4) 在创建 Pod 时, 引用该 Secret:

```
$cat >pods.yaml<<EOF
apiVersion: v1
kind: Pod
metadata:
  name: mypod2
spec:
```

```

containers:
- name: foo
  image: janedoe/awesomeapp:v1
imagePullSecrets:
- name: myregistrykey
EOF
$ kubectl create -f pods.yaml

```

其结果如图 2.14 所示。

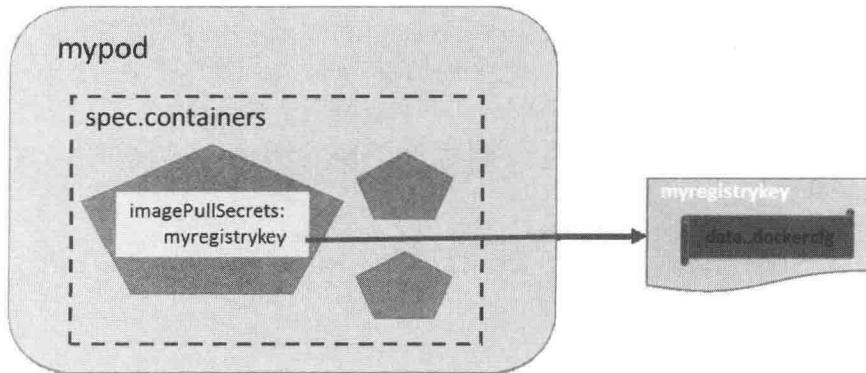


图 2.14 imagePullSecret 引用 Secret

Pod 创建时会验证所挂载的 Secret 是否真的指向一个 Secret 对象，因此 Secret 必须在任何引用它的 Pod 之前被创建。Secret 对象属于 Namespace，它们只能被同一个 Namespace 中的 Pod 所引用。

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。

Kubelet 目前只支持 Pod 使用由 API Server 创建的 Secret。Pod 包括 Kubectl 创建的 Pod 或间接被 Replication Controller 创建的 Pod，不包括 Kubelet 通过--manifest-url 参数、--config 参数或 REST API 创建的 Pod（这些都不是通用的创建 Pod 的方法）。

在使用 Mount 方式挂载 Secret 时，Container 中 Secret 的“data”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。

Secret 包含三种类型：Opaque、ServiceAccount 和 Dockercfg。在前面已经举例说明了如何创建 Opaque 和 Dockercfg 类型的 Secret。下面的例子为创建一个 Service Account Secret：

```
{
    "kind": "Secret",
    "metadata": {
        "name": "mysecret",
        "annotations": {
            "kubernetes.io/service-account.name": "myserviceaccount"
        }
    },
    "type": "kubernetes.io/service-account-token"
}
```

2.4.5 Service Account

Service Account 是多个 Secret 的集合。它包含两类 Secret：一类为普通 Secret，用于访问 API Server，也被称为 Service Account Secret；另一类为 imagePullSecret，用于下载容器镜像。如果镜像库运行在 Insecure 模式下，则该 Service Account 可以不包含 imagePullSecret。在下面的例子中创建了一个名为 build-robot 的 Service Account，并查询该 Service Account 的信息：

```
$ cat > serviceaccount.json <<EOF
{
    "kind": "ServiceAccount",
    "apiVersion": "v1",
    "metadata": {
        "name": "build-robot"
    }
}
```

```

" metadata " : {
    " name " : " myserviceaccount "
},
" secrets " : [
{
    " kind " : " Secret " ,
    " name " : " mysecret " ,
    " apiVersion " : " v1 "
},
{
    " kind " : " Secret " ,
    " name " : " mysecret1 " ,
    " apiVersion " : " v1 "
}
],
" imagePullSecrets " : [
{
    " name " : " mysecret2 "
}
]
}
EOF
$ kubectl create -f serviceaccount.json
$ kubectl get serviceaccounts build-robot -o json

```

该 Service Account 包含了对两类 Secret 的引用：一类用于下载 Image，一类用于访问 API Server。如图 2.15 所示。

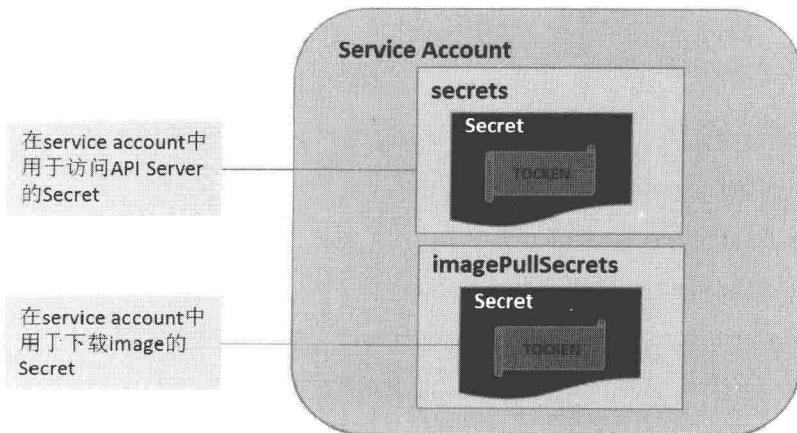


图 2.15 Service Account 中的 Secret

通过下列命令可以查询 Namespace 中的 Service Account 列表：

```
kubectl get serviceAccounts
```

Pod 和 Service Account 是如何建立关系的呢？如果在创建 Pod 时没有为 Pod 指定 Service Account，则系统会自动为其指定一个在同一命名空间(Namespace)下的名为“default”的 Service Account。如果想要为 Pod 指定其他 Service Account，则可以在 Pod 的创建过程中指定“spec.serviceAccountName”的值为相应的 Service Account 的名称。如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mycontainter
      image: nginx:v1
  serviceAccountName:myserviceaccount
```

图 2.16 列出了 Pod、Service Account 及 Secret 的关系。

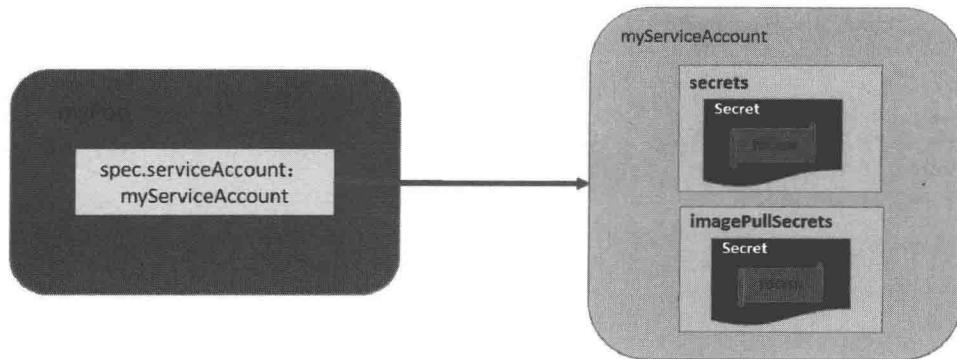


图 2.16 Pod、Service Account 和 Secret 的关系

在实现系统自动化的过程中，Service Account 会和下面三个功能一起工作：

- (1) Admission Controller;
- (2) Token Controller;
- (3) Service Account Controller。

Admission Controller 是 API Server 的一部分，在对请求进行认证和授权后，对请求做准入控制。当 API Service 的启动参数中加入如下内容时：

```
-admission_control=ServiceAccount
```

则 API Server 的 Admission Controller 会启用准入控制的 Service Account 功能。

如果 Admission Controller 启用了 Service Account 功能，则当用户在某个 Namespace（默认为 default）中创建和修改 Pod 时，Admission Controller 会做以下事情。

(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) 如果创建和修改 Pod 时 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) 添加一个“volume”给 Pod，在该“volume”中包含一个能访问 API Server 的 Token（该 Token 来自 Service Account Secret）；

(5) 通过添加“volumeSource”的方式，将上面提到的“volume”挂载到 Pod 中所有容器的 /var/run/secrets/kubernetes.io/serviceaccount 目录中。

Token Controller 和 Service AccountController 在该自动化过程中所起到的作用请参考 2.2.5 节。

2.5 网络原理

关于 Kubernetes 网络，我们通常有这些问题需要回答，如图 2.17 所示。



图 2.17 Kubernetes 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试验来将这些相关的知识串联成一个整体。

2.5.1 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 的方案能很好地利用了现有的各种域名解析和发现机制。

另外，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缺省支持的。另外，常见的其他公用云服务商如亚马逊等，在它们的公有云计算环境下也是缺省支持这个模型的。

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。2.5.5 节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。

2.5.2 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 2.18 所示。



图 2.18 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，彼此之间互相无法通信，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 2.19 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同属于它的进程组私有的，和其他进程组不冲突。

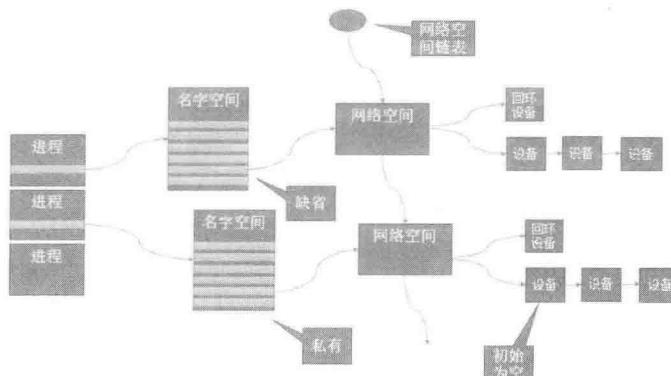


图 2.19 命名空间内核结构

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。

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，这个命令需要被 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。对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 2.20 是 Veth 设备对的示意图。

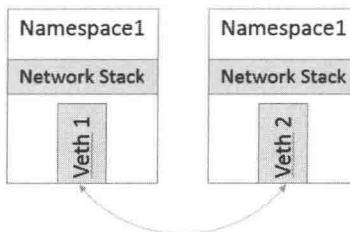


图 2.20 Veth 设备对示意图

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: eno16777736: <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: eno16777736: <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
```

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，所以我们既可以将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 2.21 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。

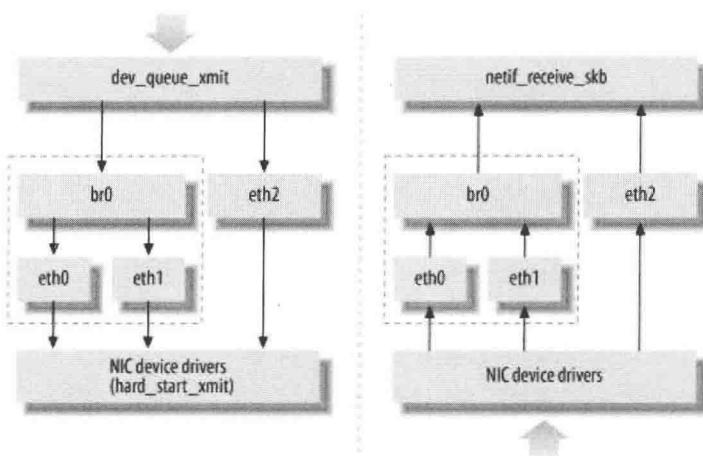


图 2.21 网桥的位置

如图 2.21 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 xxxx.xxxx.xxxx.xxxx
```

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节点，通过这些挂节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。整个挂节点技术叫作 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 2.22 中的深色椭圆所示。

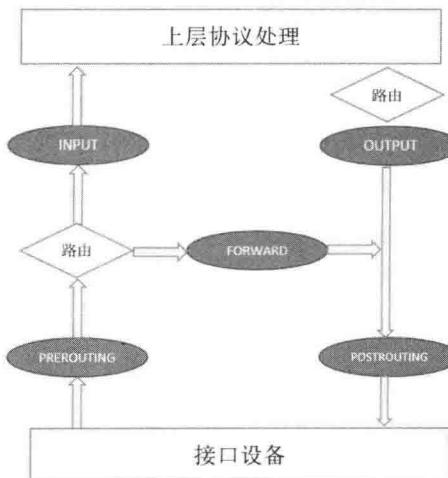


图 2.22 Netfilter 挂接点

1) 规则表 Table

这些挂节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 2.23 所示。

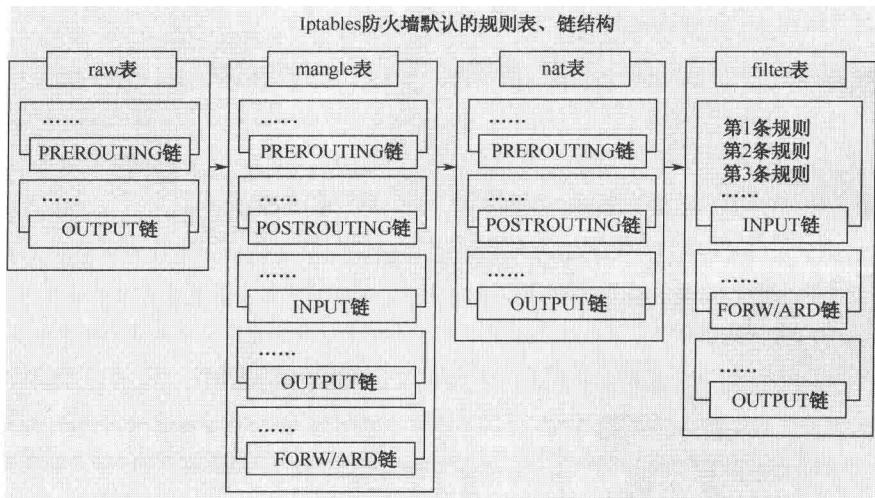


图 2.23 不同表的挂接点

当 Linux 协议栈的数据处理运行到挂节点时，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收或者拒绝。

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。

3) 匹配参数

匹配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。

4) 匹配动作

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。

5) 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。这里只介绍一下如何查看规则表，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是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，如果任何条目的第一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的匹配 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 路由表的建立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。

2.5.3 Docker的网络实现

标准的Docker支持以下4类网络模式。

- ① host模式：使用--net=host指定。
- ② container模式：使用--net=container:NAME_or_ID指定。
- ③ none模式：使用--net=none指定。
- ④ 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，缺省的名字是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 2.24 所示就是 Docker 的缺省桥接网络模型。

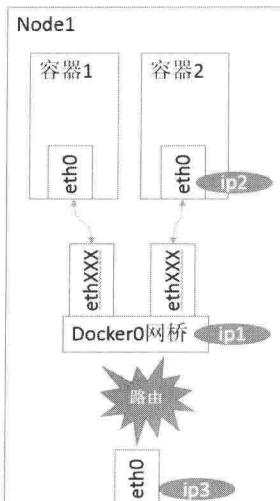


图 2.24 缺省的 Docker 网络桥接模型

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 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，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，所以在缺省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）。

为了让它们跨节点相互通信，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: eno16777736: <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 eno16777736
        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 表中有三条记录，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 表第三条的含义是，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，第一条也是一个框架，因为后继的 DOCKER 链是空的。

(4) 在 FILTER 表中，第三条是说，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，第四条是说，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，第二条是说，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 eno16777736 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 eno16777736 proto kernel scope link src 192.168.1.132
192.168.1.0/24 dev eno16777736 proto kernel scope link src 192.168.1.132
metric 100
```

可以看到：

(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 链都还是空的。

(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 24981a750ala bash
[root@24981a750ala /]# 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@24981a750ala /]# 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，并且也已经配置了地址 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被客户广为好评，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 2.25 所示是其概念图。

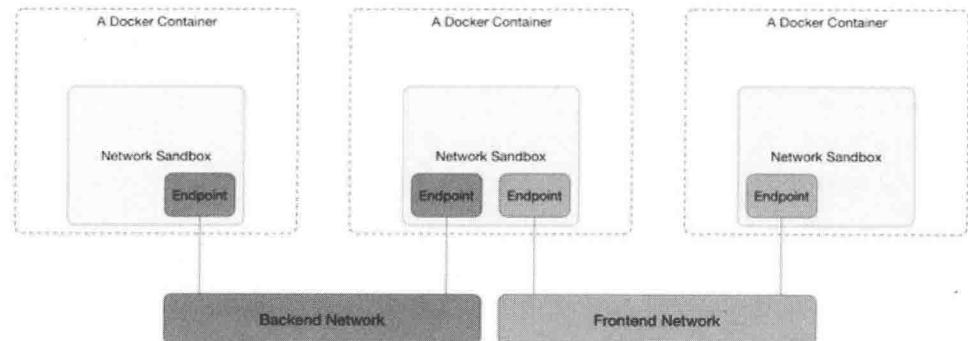


图 2.25 Libnetwork 概念图

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。

2.5.4 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) 紧密耦合的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；

接下来，我们看看Kubernetes是如何一一解决这些场景下的网络通信问题的。

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。在没有容器技术出来之前，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2.26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2.26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，容器 1 访问这个 MySQL 直接使用 localhost:3306，就能直接访问这个运行在容器 2 上的 MySQL 了。

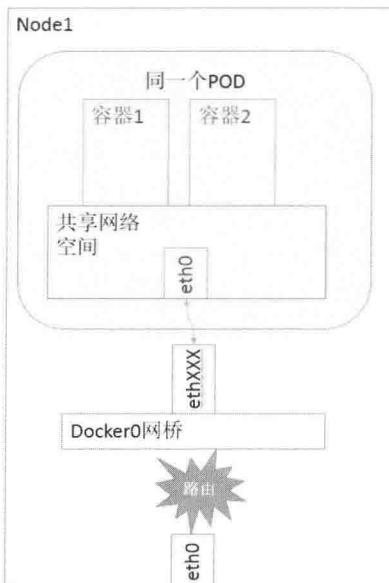


图 2.26 Kubernetes 的 Pod 网络模型

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 2.27 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。

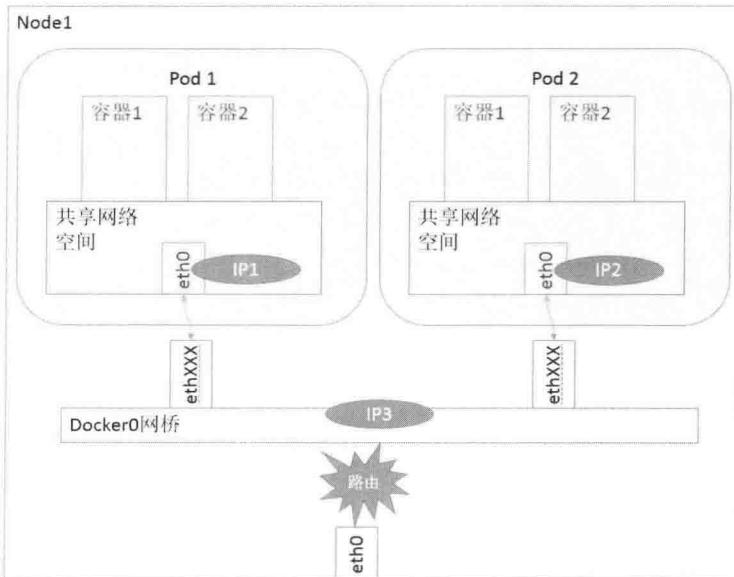


图 2.27 同一个 Node 内的 Pod 关系

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。

另外一方面，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 Fluunel 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。也就是说要先要找到 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 2.28 所示。

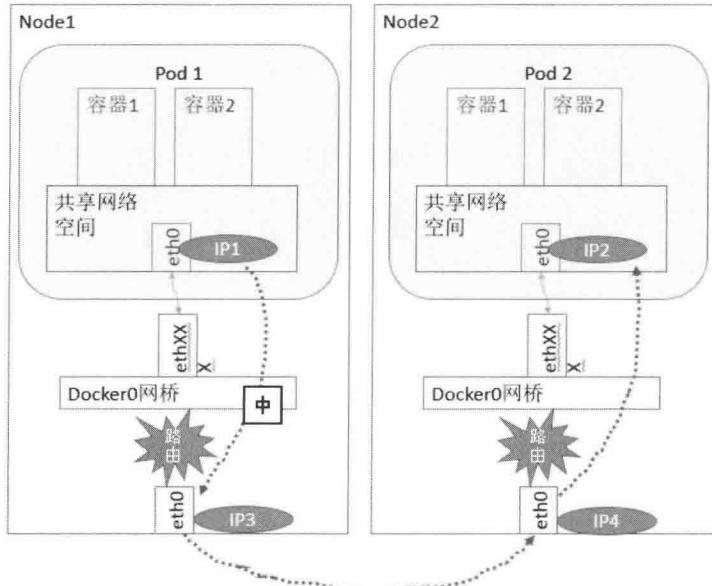


图 2.28 跨 Node 的 Pod 通信

在图 2.28 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 2.5.5 节会介绍几个常用的组件和它们的组网原理。

3. Pod 到 Service 之间的通信

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 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。kube-proxy 和后端 Pod 的通信方式与标准的 Pod 到 Pod 的通信方式完全相同。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 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，所以如果 Pod 需要访问 Service，则它所在的那个 Node 上必须运行 kube-proxy，并且在每个 Kubernetes 的 Node 上都会运行 kube-proxy 组件。在 Kubernetes 集群内部，对 Service Cluster IP 和 Port 的访问可以在任意 Node 上进行，这是因为每个 Node 上的 kube-proxy 针对该 Service 都设置了相同的转发规则。

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 2.29 所示。

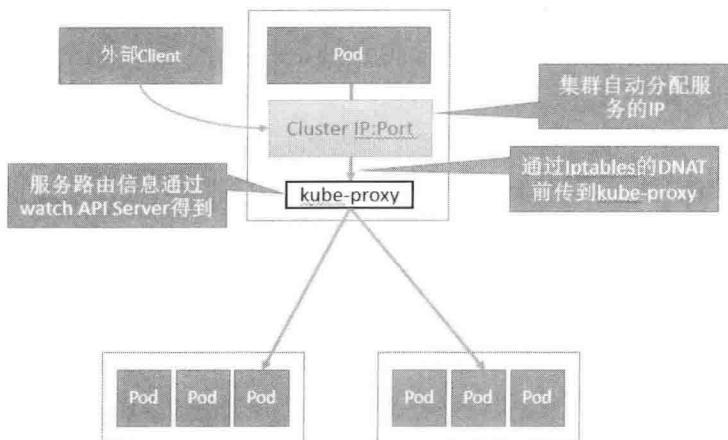


图 2.29 Service 的负载均衡转发规则

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？

首先，目前 kube-proxy 的负载均衡器只支持 ROUNDROBIN 算法。ROUNDROBIN 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 ROUNDROBIN 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 ROUNDROBIN 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。后面的请求就会粘连到这个创建好的 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 进程为每个 Service 都建立了一个“服务代理对象”，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“负载均衡器组件”，用来实现 SocketServer 上收到的连接到后端多个 Pod 连接之间的负载均衡和会话保持能力。

kube-proxy 通过查询和监听 API Server 中 Service 与 Endpoints 的变化来实现其主要功能，包括为新创建的 Service 打开一个本地代理对象(代理对象是 kube-proxy 程序内部的一种数据结构，一个 Service 端口是一个代理对象，包括一个用于监听服务请求的 SocketServer)，接收请求，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 判断本地是否存在对应的服务代理对象，如果不存在就新建，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（CREDITIRIA）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，我们以本书开始的 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-PORALS-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-PORALS-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。

4. 外部到内部的访问

Pod 作为基本的资源对象，除了会被集群内部的 Pod 访问，也会被外部使用。服务是对一组相同功能的 Pod 的抽象，以它为单位对外提供服务是最合适的粒度。

由于 Service 对象在 Cluster IP Range 池中分配到的 IP 只能在内部访问，所以其他 Pod 都可以无障碍地访问到它。但如果这个 Service 作为前端服务，准备为集群外的客户端提供服务，就需要外部能够看到它。Kubernetes 支持两种对外提供服务的 Service 的 Type 定义：NodePort 和 LoadBalancer。

1) NodePort

在定义 Service 时指定 spec.type=NodePort，并指定 spec.ports.nodePort 的值，系统就会在 Kubernetes 集群中的每个 Node 上打开一个主机上的真实端口号。这样，能够访问 Node 的客户端就能通过这个端口号访问到内部的 Service 了。

2) LoadBalancer

如果云服务商支持外接负载均衡器，则可以通过 `spec.type=LoadBalancer` 定义 Service，同时需要指定负载均衡器的 IP 地址。使用这种类型需要指定 Service 的 `nodePort` 和 `clusterIP`。

对这个 Service 的访问请求将会通过 LoadBalancer 转发到后端 Pod 上去，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。

3) 外部访问内部 Service 的原理

我们从集群外部访问集群内部，最终都是落在具体的 Pod 上。通过 NodePod 的方式，就是将 `kube-proxy` 放出去，利用 `Iptables` 为服务的 `NodePort` 设置规则，将对 Service 的访问转到 `kube-proxy` 上，这样 `kube-proxy` 就可以使用和内部 Pod 访问服务一样的方式来访问后端的一组 Pod 了。这种模式就是利用 `kube-proxy` 作为负载均衡器，处理外部到服务进一步到 Pod 的访问。

而更常用的是外部均衡器模式（LoadBalancer）。通常的实现是使用一个外部的负载均衡器（例如 GCE 的 FR 或者 AWS 的 ELB），这些均衡器面向集群内的所有节点。当网络流量发送到 LoadBalancer 地址时，它会识别出这是某个服务的一部分，然后路由到合适的后端 Pod。

所以从外面访问内部的 Pod 资源，就有了很多种不同的组合。

- (1) 外面没有负载均衡器，直接访问内部的 Pod。
- (2) 外面没有负载均衡器，直接通过访问内部的负载均衡器来访问 Pod。
- (3) 外面有负载均衡器，通过外部负载均衡器直接访问内部的 Pod。
- (4) 外面有负载均衡器，通过访问内部的负载均衡器来访问内部的 Pod。

第 1 种情况的场景十分少见，只是在特殊的时候才需要。我们在实际的生产项目中需要逐一访问启动的 Pod，给它们发送一个刷新指令。只有这种情况下才使用这种方式。这需要开发额外的程序，读取 Service 下的 Endpoint 列表，逐一和这些 Pod 进行通信。通常要避免这种通信方式，例如可以采取每个 Pod 从集中的数据源拉命令的方式，而不是采取推命令给它的方式来避免。因为具体到每个 Pod 的启停本来就是动态的，如果我们依赖了具体的 Pod，就相当于绕开了 Kubernetes 的 Service 机制，虽然能够实现，但是不理想。

第 2 种情况就是 NodePod 的方式，外部的应用直接访问 Service 的 NodePod，并通过 `kube-proxy` 这个负载均衡器访问内部的 Pod。

第 3 种情况是 LoadBalancer 模式，因为外部的 LoadBalancer 是具备 Kubernetes 知识的负载均衡器，它会去监听 Service 的创建，从而知晓后端的 Pod 启停变化，所以它有能力直接和后端的 Pod 进行通信。但是这里有个问题需要注意，那就是这个负载均衡器需要有办法直接和 Pod 进行通信。也就是说要求这个外部的负载均衡器使用和 Pod 到 Pod 一样的通信机制。

第 4 种情况也很少用，因为需要经历两级的负载均衡设备，而且网络的调用被两次随机负载均衡后，更难跟踪了。在实际生产环境中出了问题排错时，很难跟踪网络数据的流动过程。

综上所述，无论是外部的负载均衡器，还是内部的 kube-proxy 负载均衡器，都是 Service 可感知的。

4) 外部硬件负载均衡器模式

在很多实际的生产环境中，由于是在私有云环境中部署 Kubernetes 集群，所以传统的负载均衡器都对 Service 无感知。实际上我们只需要解决两个问题，就可以将它变成 Service 可感知的负载均衡器，这也是实际系统中理想的外部访问 Kubernetes 集群内部的模式。

(1) 通过写一个程序来监听 Service 的变化，将变化按照负载均衡器的通信接口，作为规则写入负载均衡器。

(2) 给负载均衡器提供直接访问 Pod 的通信手段。

我们举一个实际的例子来说明这个过程，如图 2.30 所示。

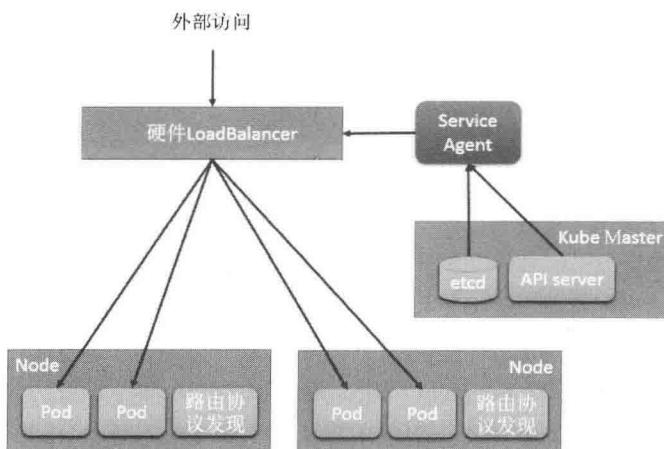


图 2.30 自定义外部负载均衡器访问 Service

这里提供了一个 Service Agent 来实现 Service 变化的感知。该 Agent 能够直接从 etcd 中或者通过接口调用 API Server 来监控 Service 及 EndPoint 的变化，并将变化写入外部的硬件负载均衡器中。

同时每台 Node 上都运行着有路由发现协议的软件，该软件负责将这个 Node 上所有的 IP 地址通过路由发现协议组播给网络内的其他主机，当然也包含硬件负载均衡器。这样硬件负载均衡器就能知道每个 Pod 实例的 IP 地址是在哪台 Node 上了。

通过上述两个步骤，就建立起一个基于硬件的外部可感知 Service 的负载均衡器，效果和 GCE 中的 LoadBalancer 一样。

2.5.5 开源的网络组件

Kubernetes的网络模型假定了所有Pod都在一个可以直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地址之间建立一个叠加网络(Overlay Network)，通过这个叠加网络，将数据包原封不动地传递到目标容器内。

通过图2.31来看看Flannel是如何实现这两点的，如图2.32所示。

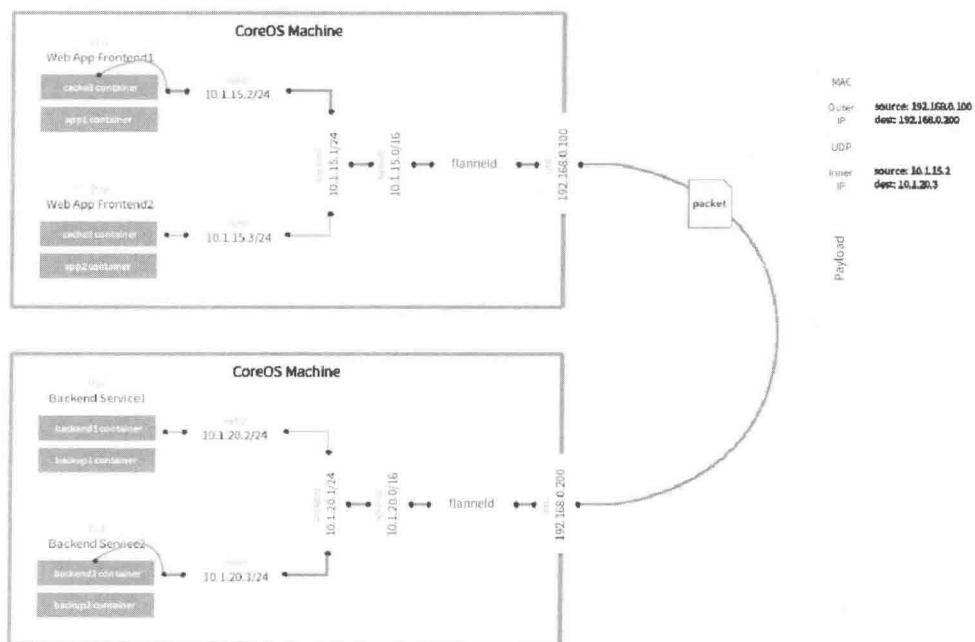


图2.31 Flannel架构图

通过图 2.31 可以看到，首先 Flannel 创建了一个 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。

我们看一下 Flannel 是如何做到使为不同 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 模型缺省地使用了 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 2.32 所示。

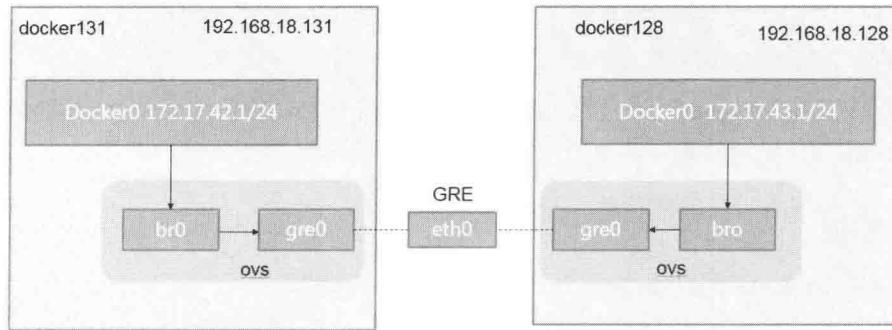


图 2.32 OVS with GRE 原理图

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 2.33 所示。

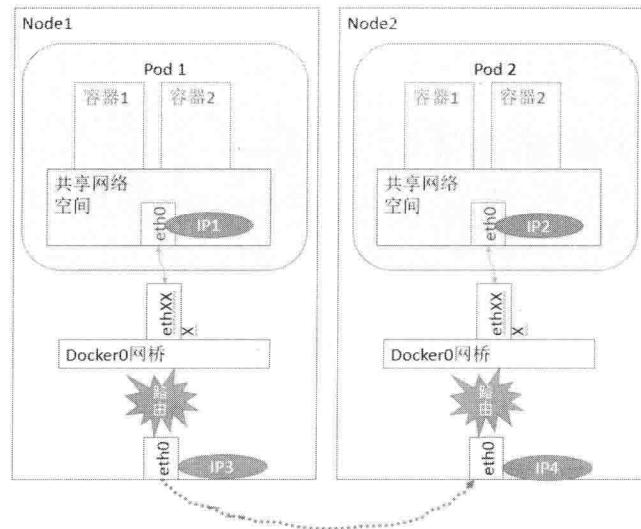


图 2.33 直接路由 Pod 到 Pod 通信

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。由于路由发现能够给网络上的所有设备接收到，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。

2.5.6 Kubernetes 网络试验

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，来看看我们的试验环境，如图 2.34 所示。

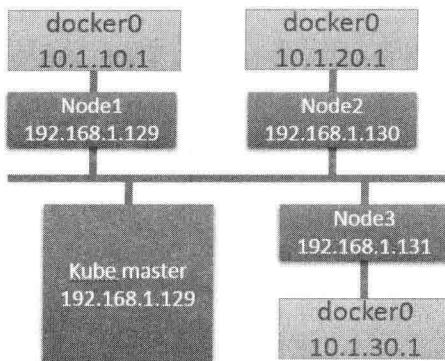


图 2.34 实验环境

Kubernetes 的网络模型要求每一个 Node 上的容器都可以相互访问。

缺省的 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）作为其缺省网关。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。如果我们将试验环境改画成一个网络图，那么它看起来如图 2.35 所示。

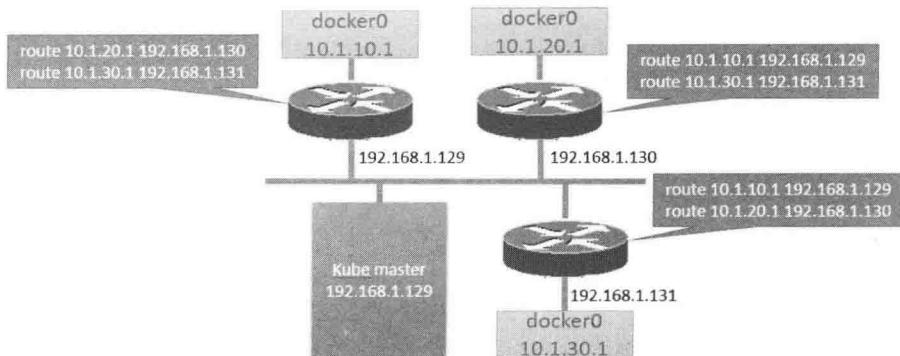


图 2.35 实验环境网络图

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）作为它的缺省网关。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。

2.5.6.1 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: 192.168.1.128:1180/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 Controllers、Pods 和其他 Services：

```

# 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:fe97:99 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-4ol1g 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 上的 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-n9nlm_
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 缺省的网络模型 bridge；而我们检查的第二个容器，也就是在我们 RC/Pod 中定义运行的 php-redis 容器，使用了非缺省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一个 Pod 中的容器，而将第二个 Pod 内的容器关联到第一个容器呢？我们认为 Kubernetes 从两个方面来考虑这个问题：第一，如果 Pod 有超过两个容器的话，则连接这些容器可能不容易，第二，后面的容器还要依赖第一个被关联的容器，如果第二个容器关联到第一个容器，且第一个容器死掉的话，第二个也将死掉。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 2.36。

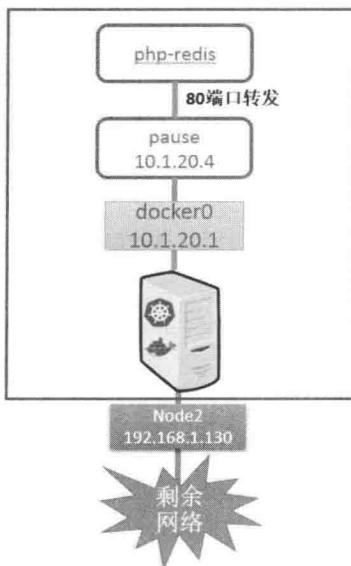


图 2.36 启动 Pod 后网络模型

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 2.36 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了 `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 的节点提供一些缺省的服务，上面的规则就是 Kubernetes 的缺省服务需要的。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。

2.5.6.2 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 Kubemaster 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缺省网关（也就是 docker0 网桥）。在继续实验前，先登录 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
.....
```

第一行是挂在 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。第二行是挂在 OUTPUT 链上的目标地址 NAT，做了和上述第一行规则类似的工作，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-x5dwv  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 2.37 所示。

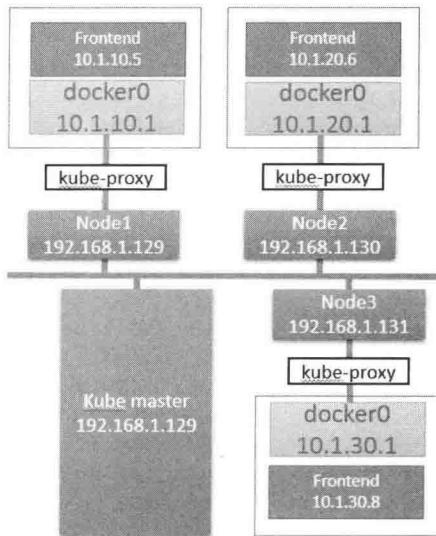


图 2.37 启动服务后的结构

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。

再打开第一个窗口中运行第二个 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，在第二个窗口中运行 tcpdump 的命令。

```
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。这时再启动第三个窗口，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一个窗口中网络抓包信息的含义（物理网卡上的网络流量），并用虚线标出第二个窗口中网络抓包信息的含义（docker0 网桥上的网络流量），如图 2.38 所示。

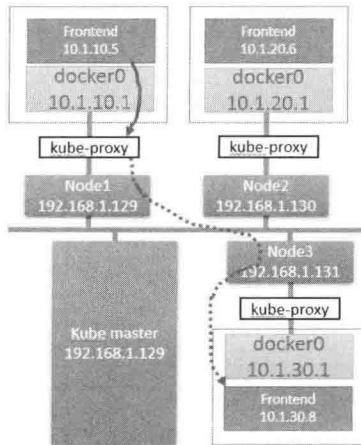


图 2.38 数据流动情况图 1

注意，图 2.38 中，虚线是绕过了 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。

在查看第二个捕获包的窗口时，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 2.39 所示。

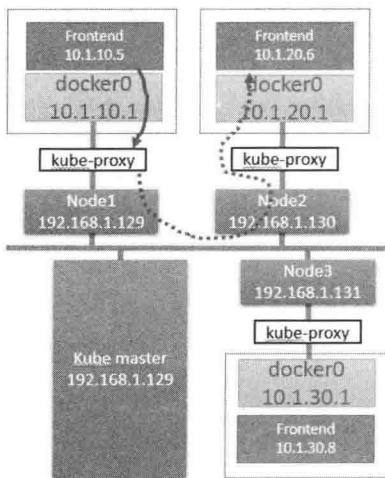


图 2.39 数据流动情况图 2

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。关于服务的最重要的概念就是它使得我们可以快速、容易地水平扩展部署 Pod。结合使用 Replication Controller 对 Pod 的复制部署，可以看出这是功能很强大的特性。

第3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。下面就让我们从 REST 开始说起。

3.1 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这两个关注点，使得不同的用户终端共享相同的数据成为了可能。

(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。

3.2 Kubernetes API 详解

3.2.1 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 3.1 所示。



图 3.1 swagger-ui

单击 `api/v1` 可以查看所有 API 的列表，如图 3.2 所示。



图 3.2 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 3.3 所示。



图 3.3 Create a Pod API

单击链接展开, 即可查看详细的 API 接口说明, 如图 3.4 所示。

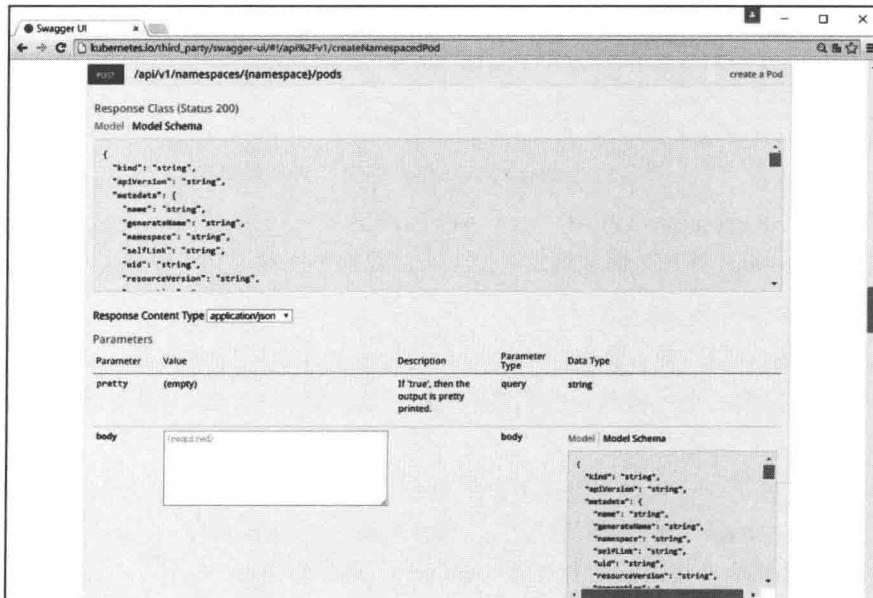
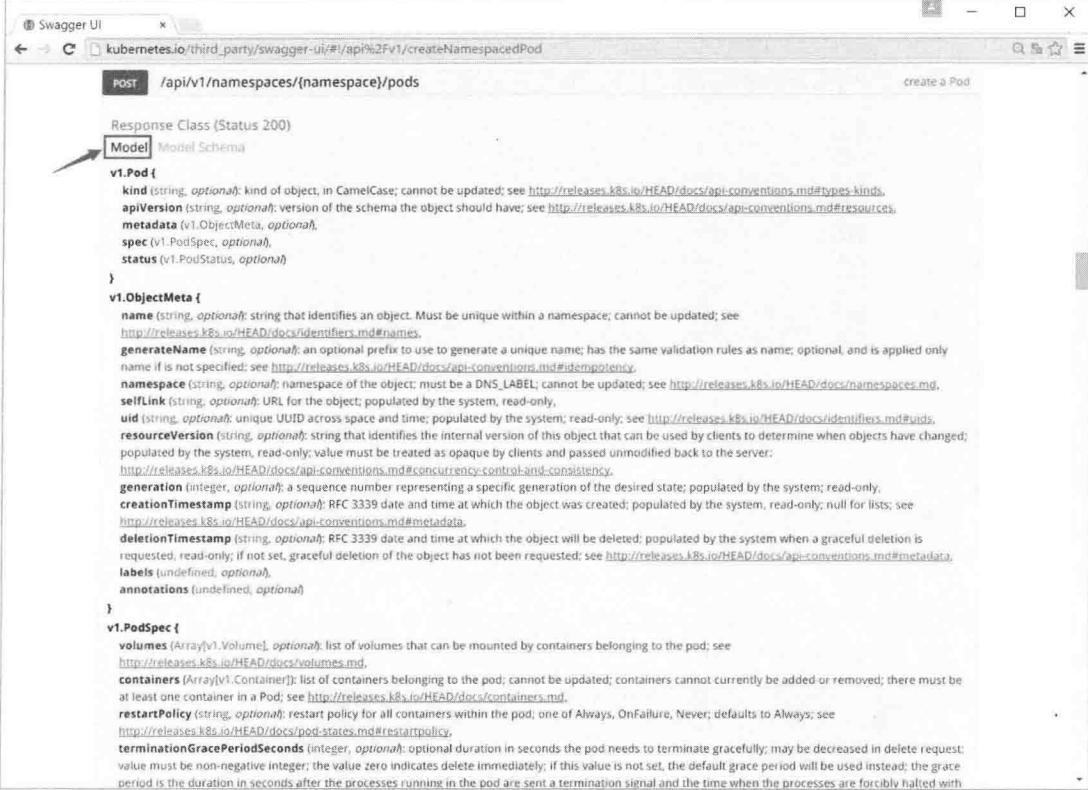


图 3.4 Create a Pod API 详细说明

单击 Model 链接，则可以查看文本格式显示的 API 接口描述，如图 3.5 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#name.
  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 3.5 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做创建、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。

3.2.2 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。

3.2.3 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，删除选项 (DeleteOptions) 中可以指定的优雅删除 (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。

3.2.4 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 3.1 列出了 API Server 可能返回的状态码。

表 3.1 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

续表

状态码	编码	描述
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
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",
" 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。

3.3 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，一个为 OSGI，另一个为 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。

3.3.1 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 Services 及它们的客户端的流程，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 Services 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 Services。因此此处选用 Jersey 框架开发 RESTful Web Services 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 3.6 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 3.6 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。第一个定义

的对象为 ResourceType，它定义了访问资源的类型；第二个定义的对象是 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；第三个对象是 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(", ");
        }
    }
    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 : inLabels.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 3.2 所示。

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.imp.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。

3.3.2 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 3.7 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 3.7 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 watcher 方式访问资源做了很好的支持。

3.3.3 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 3.3 列出了各资源对象的基本 API 接口。

表 3.3 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	获得 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	

续表

资源类型	方法	URL Path	说明	备注
Service	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 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	
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	获得某个 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	

续表

资源类型	方法	URL Path	说明	备注
	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
	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	

续表

资源类型	方法	URL Path	说明	备注
	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	
	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	
EVENTS	PUT	/api/v1/namespaces/{namespace}/secrets/{name}	替换某个 Namespace 下的一个 Secret 对象	
	GET	/api/v1/events	获取 Event 列表	
	POST	/api/v1/events	创建一个 Event 对象	

续表

资源类型	方法	URL Path	说明	备注
EVENTS	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	
COMPONENTSTATUSSES	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	
	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	
RESOURCEQUOTAS	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	

续表

资源类型	方法	URL Path	说明	备注
	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
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	

续表

资源类型	方法	URL Path	说明	备注
	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	
	PUT	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}	替换某个 Namespace 下的一个 PersistentVolumeClaim 对象	
	PUT	/api/v1/namespaces/{namespace}/persistentvolumeclaims/{name}/status	替换某个 Namespace 下的一个 PersistentVolumeClaim 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 Jersey 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": "192.168.1.128:1180/kubernetes/example-guestbook-php-redis"
            }
        ]
    }
}
```

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": "192.168.1.128:1180/kubernetes/example-guestbook-php-redis"
            }
        ]
    }
}
```

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": "192.168.1.128:1180/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": "192.168.1.128:1180/kubernetes/example-guestbook-php-redis",
                "imageID": "docker://5630952871a38cddffda9ec611f5978ab0933628fc54cd7d7677ce6b17de33f",
                "containerID": "docker://7bf0d454c367418348711556e667fd1ef6a04d7153d24bfac2e2e06da634a9f"
            }
        ]
    }
}

```

接下来的两个例子实现了 3.2.4 节中提到的两种 Merge 方式：Merge Patch 和 Strategic Merge Patch。

第一种 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"  
    }  
}  
}
```

第二种 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": "192.168.1.128:1180/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": "192.168.1.128:1180/kubernetes/example-guestbook-php-redis" ,
            "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": ""
        }
    ],
    "imageID": "docker://5630952871a38cddffda9ec611f5978ab0933628fc54cd
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 3.4 所示。

表 3.4 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

续表

资源类型	类别	方法	URL Path	说明
		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
		GET	/api/v1/watch/namespaces/{namespace}/services/{name}	监听某个 Service 的变化
	PROXY	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

续表

资源类型	类别	方法	URL Path	说明
		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
		GET	/api/v1/watch/namespaces/{namespace}/pods/{name}	监听某个 Pod 的变化
	PROXY	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

续表

资源类型	类别	方法	URL Path	说明
		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
		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
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

续表

资源类型	类别	方法	URL Path	说明
		GET	/api/v1/watch/namespaces/{namespace}/endpoints/{name}	监听某个 Endpoint 的变化
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
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
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
		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

续表

资源类型	类别	方法	URL Path	说明
		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 testProxyPod() {
    //访问第一种 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);
```

```
//访问第二种 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4章

Kubernetes 运维指南

本章将对 Kubernetes 系统中需要配置的系统参数、配置文件、运维技巧、资源配额管理、网络配置、系统监控及 Trouble Shooting 等方面进行详细说明，通过实际操作的案例对 Kubernetes 的运维工作提供指导。

4.1 Kubernetes 核心服务配置详解

在第 1 章安装内容部分对 Kubernetes 各服务启动进程的关键配置参数进行了简要说明，而实际上 Kubernetes 的每个服务都提供了许多可配置的参数。这些参数涉及了安全性、性能优化及功能扩展（Plugin）等方面。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。

4.1.1 基础公共配置参数

基础公共配置参数适用于所有服务，如表 4.1 所示包括 kube-apiserver、kube-controller-

manager、kube-scheduler、Kubelet、kube-proxy 等。在本节会进行统一说明，不再在每个服务的参数列表中列出。

表 4.1 基础公共配置参数表

参数名和取值示例	说 明
--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	输出到 stderr，不输出到日志文件
--alsologtostderr=false	如果设置为 true，将日志输出到文件同时也输出到 stderr
--stderrthreshold=2	在该 threshold 级别之上的日志将输出到 stderr
--v=0	glog 日志级别
--vmodule=	glog 基于模块的详细日志级别
--version=false	打印版本信息然后退出

4.1.2 kube-apiserver

kube-apiserver 的参数见表 4.2

表 4.2 kube-apiserver 的参数表

参数名和取值示例	说 明
--address=127.0.0.1	旧版本参数，已被--insecure-bind-address 替代
--port=8080	旧版本参数，已被--insecure-port 替代
--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	提供不安全不认证访问的监听端口，默认为 8080。假设该端口号在防火墙进行了配置，以使得外部客户端无法直接访问
--bind-address=0.0.0.0	Kubernetes API Server 在本地址的 6443 端口开启安全的 HTTPS 服务，默认为 0.0.0.0
--secure-port=6443	使用 HTTPS 的端口号，设置为 0 表示不启用 HTTPS
--public-address-override=0.0.0.0	旧版本参数，已被“--bind-address”替代
--advertise-address=<nil>	用于广播给集群所有成员自己的 IP 地址，不指定系统使用“--bind-address”定义的 IP 地址
--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

续表

参数名和取值示例	说 明
--admission-control = "AlwaysAdmit"	<p>加载的“准入控制器”参数，每个准入控制器是一段二进制代码，以插件方式动态加载，参照值是以逗号分隔的多个准入控制权，形成一个控制链，若控制链上的任何一个节点不通过，则 API Server 拒绝此调用请求。目前可用的准入控制权如下</p> <ul style="list-style-type: none"> ● AlwaysAdmit：允许所有请求。 ● AlwaysDeny：禁止所有请求，多用于测试环境。 ● NamespaceLifecycle：它会观察所有请求，如果请求尝试创建一个不存在的 namespace，则这个请求被拒绝。 ● 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，如果你想要使用 ServiceAccount 对象，那么强烈推荐你使用它。 ● SecurityContextDeny：这个插件将会使用了 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。 <p>另外，如果启用多种准入选项，则建议的加载顺序是：</p> <pre>--admission-control=NamespaceLifecycle,LimitRanger,SecurityContextDeny,ServiceAccount,ResourceQuota</pre>
--admission-control-config-file= " "	与准入控制相关的配置文件，一般不使用配置文件，而是通过 API Server（或 Kubectl）来维护相关的准入配置约束（规则）
--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
--api-burst=200	用于服务限流，默认最大突发流量为每秒 200 次调用
--api-prefix= "/api "	API 访问请求前缀，默认为 "/api"
--api-rate=10	用于服务限流所允许的每秒查询 API 调用的次数
--authorization-mode= "AlwaysAllow"	安全端口上的认证方式，可选值包括：AlwaysAllow、AlwaysDeny、ABAC
--authorization-policy-file= " "	当--authorization-mode 设置为 ABAC 时使用的 csv 格式的认证策略文件
--basic-auth-file= " "	如果指定，则这个文件将被用于通过 HTTP 基本认证的方式访问 apiserver 的安全端口
--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

续表

参数名和取值示例	说 明
--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
--cloud-provider= " "	云服务商的名称
--cluster-name= " kubernetes "	Kubernetes 集群名，也表现为实例名的前缀
--cors-allowed-origins=[]	CORS（跨域资源共享）设置允许访问的源域列表，用逗号进行分隔，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
--etcd-config= " "	etcd 客户端配置文件路径，与--etcd-servers 互斥
--etcd-prefix= " /registry "	etcd 中所有资源路径的前缀，默认为 “/registry”
--etcd-servers=[]	以逗号分隔的 etcd 服务列表，与--etcd-config 互斥
--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
--external-hostname= " "	此主机名为外部能识别的本节点的名字，用来生成供外部系统访问的 API URL，其中一个用途就是在 Swagger API 的 Docs 文档中使用此主机名
--httptest.serve=	如果提供了此参数，则 Kubernetes API Server 会在此地址上开启一个 HTTP Server，这个 Server 提供了部分 API 的 Mock 实现（主要是与 Pod 相关的 API），用于测试
--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 的客户端秘钥文件路径
--kubelet-https=true	指定 Kubelet 是否使用 HTTPS 连接
--kubelet-port=10250	Kubelet 监听的端口号
--kubelet-timeout=5s	Kubelet 执行操作的超时时间
--long-running-request-regexp= " (^)((watch proxy)(/ \$) (logs portforward exec) /?\$)"	用于对需要长时间运行的请求不进行请求数量限制，用正则表达式表示
--master-service-namespace= " default "	Kubernetes Master 服务的命名空间
--max-requests-inflight=400	同时处理的最大请求数量，超过这个数量的请求将被拒绝。设置为 0 表示不限制数量
--min-request-timeout=1800	最小请求处理超时时间，单位为秒
--old-etcd-prefix= " "	指定以前 etcd 中资源路径的前缀
--runtime-config=	一组 key=value 用于运行时的配置信息。api/<version> 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 API
--service-account-key-file= " "	包含 EM-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

续表

参数名和取值示例	说 明
--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(VIP)池, 例如 10.254.0.0/16, 这个 IP 地址池不能在 Kubernetes 所在的网络内被使用
--service-node-port-range=	Service 的 NodePort 所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 使用指定的密钥文件对 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-version= " "	存储资源时使用的版本号, 默认为系统内部使用
--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= " "	用于访问加密端口的 token 认证文件路径

4.1.3 kube-controller-manager

kube-controller-manager 的参数表见表 4.3。

表 4.3 kube-controller-manager 的参数表

参数名和取值示例	说 明
--address=127.0.0.1	绑定主机 IP 地址, 设置为 0.0.0.0 表示使用全部网络接口
--port=10252	controller-manager 监听的主机端口号, 默认为 10252
--allocate-node-cidrs=false	设置为 true 表示使用云服务商为 Pod 分配的 CIDRs, 只在被托管的公有云中有效
--cloud-config= " "	云服务商的配置文件路径
--cloud-provider= " "	云服务商的名称
--cluster-cidr=<nil>	可用 CIDR 范围
--cluster-name= " kubernetes "	Kubernetes 集群名, 也表现为实例名的前缀
--concurrent-endpoint-syncs=5	并发执行 Endpoint 同步操作的协程数, 值越大表示更快的同步操作, 但将会消耗更多 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
--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
--httptest.serve=	(参见 kube-apiserver 中的参数说明)
--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
--master= " "	Kubernetes Master apiserver 地址

续表

参数名和取值示例	说 明
--namespace-sync-period=5m0s	命名空间更新的同步时间间隔
--node-monitor-grace-period=40s	监控 Node 状态的时间间隔，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 Node 状态时间间隔（参数--node-status-update-frequency=10s）的 N 倍，N 表示 Kubelet 状态汇报的重试次数
--node-monitor-period=5s	同步 NodeStatus 的时间间隔
--node-startup-grace-period=1m0s	Node 启动的最长允许时间，超过此时间无响应则会标记 Node 为不可用状态（启动失败）
--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
--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
--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
--pvclaimbinder-sync-period=10s	同步 PV 和 PVC（容器声明的 PV）的时间间隔
--register-retry-count=10	Node 信息注册重试次数，默认为 10 次。重试的时间间隔为参数--node-sync-period 定义的值
--resource-quota-sync-period=10s	配额（Quota）使用信息同步的时间间隔，默认为 10 秒
--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

4.1.4 kube-scheduler

kube-scheduler 的参数表见表 4.4。

表 4.4 kube-scheduler 的参数表

参数名和取值示例	说 明
--address=127.0.0.1	绑定主机 IP 地址，设置为 0.0.0.0 表示使用全部网络接口
--port=10251	scheduler 监听的主机端口号，默认为 10251
--algorithm-provider= "DefaultProvider"	调度策略，默认为 DefaultProvider
--kubeconfig= " "	(参见 kube-controller-manager 中的参数说明)
--master= " "	(参见 kube-controller-manager 中的参数说明)
--policy-config-file= " "	调度策略（scheduler policy）配置文件路径
--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

4.1.5 Kubelet

Kubelet 的参数表见表 4.5。

表 4.5 Kubelet 的参数表

参数名和取值示例	说 明
--address=0.0.0.0	绑定主机 IP 地址, 设置为 0.0.0.0 表示使用全部网络接口
--enable-server=false	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, 对于开发定制自己的 Web 管理系统来说, 这个选项很有价值
--enable-debugging-handlers=false	如果为 true, 则在 http rest server 上添加 debug handlers, debug handlers 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
--port=10250	Kubelet 上的 HTTP REST Server 的端口号
--read-only-port=10255	如果设置为非零参数, 则 Kubelet 会启动一个提供“只读”操作的 HTTP REST Server
--cadvisor-port=4194	本地 cAdvisor endpoint 的端口号
--log-cadvisor-usage=false	是否记录 cAdvisor 容器的使用情况
--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 表示使用所有网络接口
--healthz-port=10248	healthz 服务监听的主机端口号, 默认为 10248
--allow-privileged=false	是否允许以特权模式启动容器, 默认为 false
--api-servers=[]	Master apiserver 地址列表, 以 ip:port 格式表示, 以逗号分隔
--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
--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= " "	云服务商的名称
--cluster-dns=<nil>	集群 DNS 服务的 IP 地址
--cluster-domain= " "	集群域名
--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
--docker-endpoint= " unix:///var/run/docker.sock "	Docker endpoint 地址
--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

续表

参数名和取值示例	说 明
--docker-root=/var/lib/docker	Docker state 根目录全路径，默认为 /var/lib/docker
--docker-run=/var/run/docker	Docker 运行时根目录的全路径，默认为 /var/lib/docker
--event-storage-age-limit=default=24h	事件保存时间列表，以 key=value 的格式表示，以逗号分隔，事件类型包括 creation、oom 等，“default” 表示所有未指定事件的类型
--event-storage-event-limit=default=1000 00	每种类型事件数量限制，以 key=value 格式表示，逗号分隔，事件类型包括 creation、oom 等，“default” 表示所有未指定事件的类型
--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
--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
--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
--sync-frequency=10s	当前正在运行的容器和其定义配置信息进行同步的最大时间间隔
--google-json-key= " "	谷歌云平台 Service Account 的 json key 文件路径
--host-network-sources= " file "	以逗号分隔的字符串参数，控制 Pod 是否被允许使用主机的网络（net=HOST），默认情况下只有来源于 File Source 源的 Pod 才能使用主机网络，如果允许所有 Pod 使用主机网络，则可以设置为 “*”
--hostname-override= " "	如果设置该项，则用作主机名，不再使用主机真实的 hostname
--httptest.serve=	参见 kube-apiserver 中的参数
--image-gc-high-threshold=90	默认为 90%，它与 image-gc-low-threshold 一起，用于设置镜像占用磁盘空间比例的高低水位，超过高水位就会清理不用的镜像，释放磁盘空间
--image-gc-low-threshold=80	参见 image-gc-high-threshold
--kubeconfig=/var/lib/kubelet/kubeconfig	参见 kube-controller-manager
--low-diskspace-threshold-mb=256	创建 Pod 所需剩余磁盘空间的下限，单位为 MB。当剩余磁盘空间低于设定值时，Kubelet 将拒绝创建新的 Pod，默认值为 256MB。
--master-service-namespace= " default "	Master 服务的命名空间
--max-pods=40	在此 Kubelet 节点上可运行的 Pod 上限
--maximum-dead-containers=100	在系统中保存的已停止的容器实例的最大数量，由于停止的容器也会消耗磁盘空间，所以超过上限以后，Kubelet 会自动清理这些容器以释放磁盘空间
--maximum-dead-containers-per-containe r=2	系统中允许的每个容器能保留的停止实例的上限数，参照 maximum-dead-containers
--minimum-container-ttl-duration=1m0s	已停止运行的容器实例在被清理释放之前的最小存活时间，例如“300ms”，“10s”或“2h45m”，超过此存活时间的实例如果满足清理条件，则会被自动清理掉
--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。警告：此参数还不具备在生产环境中使用的条件，目前仅供测试(Alpha 版本的测试功能)

续表

参数名和取值示例	说 明
--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。
--oom-score-adj=-900	Kubelet 进程的 oom_score_adj 参数值，有效范围为 [-1000, 1000]
--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 apiserver 中获取 CIDR 设置
--pod-infra-container-image="gcr.io/google_containers/pause:0.8.0"	Kubernetes 基础 pause 的镜像名称，默认从 gcr.io 下载，如果“被墙”，则此参数为解决问题的关键手段，即从别处获取此镜像并存入私有 Registry 中
--really-crash-for-testing=false	在 panics 发生时崩溃，仅用于测试
--register-node=true	注册自己的信息到 apiserver，需要设定--api-servers
--registry-qps=0	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
--registry-burst=10	默认值为 10，表示最多只能同时拉取 10 个镜像，参照 registry-qps
--resource-container="/kubelet"	当此参数不为空时，Kubelet 会运行以此命名的容器（cgroup），默认为 /kubelet，这是出于审计或者资源隔离的目的而设计的功能
--system-container=" "	可选参数，用于将非 kernel 的不在容器中的其他进程放入此容器（cgroup）中，为空表示不创建容器，此参数修改后如果要回滚到原来状态，则需要重启系统
--root-dir="/var/lib/kubelet"	Kubelet 运行时的文件存放目录
--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
--streaming-connection-idle-timeout=0	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
--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

4.1.6 kube-proxy

kube-proxy 的参数表参见表 4.6。

表 4.6 kube-proxy 的参数表

参数名和取值示例	说 明
--bind-address=0.0.0.0	主机绑定的 IP 地址，默认为 0.0.0.0
--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 表示使用所有网络接口

续表

参数名和取值示例	说 明
--healthz-port=10249	healthz 服务监听的主机端口号，默认为 10249
--kubeconfig= " "	包含 Master 地址和认证信息的配置文件路径
--master= " "	Kubernetes Master apiserver 的地址
--oom-score-adj=-899	kube-proxy 进程的 oom_score_adj 参数值，有效范围为 [-1000, 1000]
--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
--resource-container= " /kube-proxy "	参照 Kubelet 的 resource-container 参数的意义和作用

4.2 关键对象定义文件详解

本节对用户需要定义的 Pod、RC 和 Service 的配置文件进行详细说明。在模板中列出的属性为最常用的内容，完整的属性列表可以参考 API 文档中的说明。

4.2.1 Pod 定义文件详解

Pod 的定义模板（yaml 格式）如下：

```

apiVersion: v1          // Required
kind: Pod               // Required
metadata:              // Required
  name: string          // Required
  namespace: string      // Required
  labels:
    - name: string
  annotations:
    - name: string
spec:                   // Required
  containers:           // Required
    - name: string        // Required
      image: string        // Required
      imagePullPolicy: [Always | Never | IfNotPresent]
      command: [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
volumes:
- name: string
  # Either emptyDir for an empty directory
  emptyDir: {}
  # Or hostPath for a pre-existing directory on the host
  hostPath:
    path: string
restartPolicy: [Always | Never | OnFailure]
dnsPolicy: [Default | ClusterFirst]          // Required
nodeSelector: object
imagePullSecrets: object

```

对各属性的详细说明如表 4.7 所示。

表 4.7 对 Pod 定义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	v1
kind	String	Required	Pod
metadata	Object	Required	元数据
metadata.name	String	Required	Pod 名称, 需符合 RFC 1035 规范
metadata.namespace	String	Required	命名空间, 在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.containers[]	List	Required	Pod 中运行的容器的列表
spec.containers[].name	String	Required	容器名称, 需符合 RFC 1035 规范
spec.containers[].image	String	Required	容器的镜像名, 在 Node 上如果不存在该镜像, 则 Kubelet 会先下载

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：表示每次都下载镜像。 IfNotPresent：表示如果本地有该镜像，就使用本地的镜像。 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 CMD 命令
spec.containers[].workingDir	String		容器的工作目录
spec.containers[].volumeMounts[]	List		可供容器使用的共享存储卷列表
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
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		环境变量名称
spec.containers[].env[].value	String		环境变量的值
spec.containers[].resources	Object		资源限制条件
spec.containers[].resources.limits	Object		资源限制条件
spec.containers[].resources.limits.cpu	String		CPU 限制条件，将用于 docker run --cpu-shares 参数
spec.containers[].resources.limits.memory	String		内存限制条件，将用于 docker run --memory 参数
spec.volumes[]	List		在该 Pod 上定义的共享存储卷列表
spec.volumes[].name	string		共享存储卷名称，需唯一，符合 RFC 1035 规范。容器定义部分 containers[].volumeMounts[].name 将引用该共享存储卷的名称
spec.volumes[].emptyDir	Object		默认的存储卷类型，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: {}。该类型与 hostPath 类型互斥，应只定义一种

续表

属性名称	取值类型	是否必选	取值说明
spec.volumes[].hostPath	Object		使用 Pod 所在主机的目录，通过 volumes[].hostPath.path 指定。 该类型与 emptyDir 类型互斥，应只定义一种。
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.dnsPolicy	String	Required	DNS 策略，可选值包括：Default、ClusterFirst
spec.restartPolicy	Object		该 Pod 内容器的重启策略，可选值为 Always、OnFailure，默认值为 Always。 Always：容器一旦终止运行，无论容器是如何终止的，Kubelet 都将重启它。 OnFailure：只有容器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：容器终止后，Kubelet 将退出码报告给 Master，不再重启它
spec.nodeSelector	Object		指定需要调度到的 Node 的 Label，以 key=value 格式指定
spec.imagePullSecrets	Object		Pull 镜像时使用的 secret 名称，以 name=secretkey 格式定义

4.2.2 RC 定义文件详解

RC（ReplicationController）定义文件模板（yaml 格式）如下：

```

apiVersion: v1           // Required
kind: ReplicationController // Required
metadata:                // Required
  name: string            // Required
  namespace: string        // Required
  labels:
    - name: string
  annotations:
    - name: string
spec:                     // Required
  replicas: number         // Required
  selector: []             // Required
  template: object          // Required

```

对各属性的说明如表 4.8 所示。

表 4.8 对 RC 之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	ReplicationController
metadata	object	Required	元数据
metadata.name	string	Required	ReplicationController 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.replicas	number	Required	Pod 副本数量, 设置为 0 表示不创建 Pod
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.template	object	Required	容器的定义, 与 Pod 的 spec 内容相同, 参见 4.2.1 节的描述

4.2.3 Service 定义文件详解

Service 的定义文件模板 (yaml 格式) 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
  port: int
  targetPort: int
  protocol: string
status:
  loadBalancer:
    ingress:
      ip: string

```

```
hostname: string
```

对各属性的说明如表 4.9 所示。

表 4.9 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
spec.clusterIP	string		虚拟服务 IP 地址，当 type=ClusterIP 时，如果不指定，则系统将自动分配；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 需要暴露的端口号列表
spec.ports[].name	string		端口名称
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[].protocol	string		端口协议，支持 TCP 和 UDP，默认为 TCP

续表

属性名称	取值类型	是否必选	取值说明
Status	object		当 spec.type=LoadBalancer 时，设置外部负载均衡器的地址
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

4.3 常用运维技巧集锦

本节对常用的 Kubernetes 系统运维操作和技巧进行详细说明。

4.3.1 Node 的隔离和恢复

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: kubernetes-minion1
  labels:
    kubernetes.io/hostname: kubernetes-minion1
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
nodes/kubernetes-minion1
```

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           LABELS
kubernetes-minion1   kubernetes.io/hostname=kubernetes-minion1   Ready,
SchedulingDisabled
```

对于后续创建的 Pod，系统将不会再向该 Node 进行调度。

另一种方法是不使用配置文件，直接使用 kubectl patch 命令完成：

```
$ kubectl patch node kubernetes-minion1 -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。

4.3.2 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，对于一个新 Node 的加入是非常简单的。可以在 Node 节点上安装 Docker、Kubelet 和 kube-proxy 服务，然后将 Kubelet 和 kube-proxy 的启动参数中的 Master URL 指定为当前 Kubernetes 集群 Master 的地址，最后启动这些服务。基于 Kubelet 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 4.1 所示。

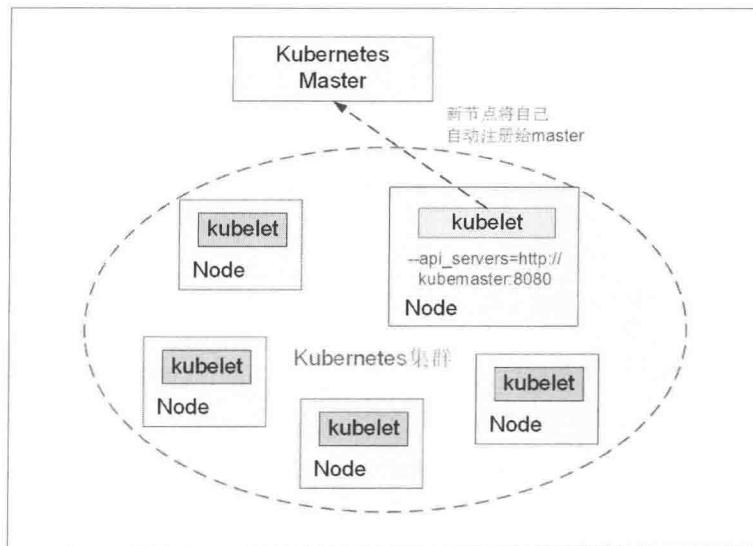


图 4.1 新节点自动注册完成扩容

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 实现了集群的扩容。

4.3.3 Pod 动态扩容和缩放

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的场景。此时我们可以利用命令 `kubectl scale rc` 来完成这些任务。以 `redis-slave` RC 为例，已定义的最初副本数量为 2，通过执行下面的命令将 `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
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，即可实现应用集群缩容：

```
$ kubectl scale rc redis-slave --replicas=1
scaled
```

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
redis-slave-4na2n  1/1     Running   0          1h
```

4.3.4 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，在已创建的对象上，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
```

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，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-
```

修改一个 Label 的值，需要加上--overwrite 参数：

```
$ kubectl label pod redis-master-bobr0 role=master --overwrite
```

4.3.5 将 Pod 调度到指定的 Node

我们知道，Kubernetes 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，我们无法知道 Pod 最终会被调度到哪个节点上。有时我们可能需要将 Pod 调度到一个指定的 Node 上，此时，我们可以通过 Node 的标签（Label）和 Pod 的 nodeSelector 属性相匹配，来达到上述目的。

首先，我们可以通过 kubectl label 命令给目标 Node 打上一个特定的标签，下面是此命令的完整用法：

```
kubectl label nodes <node-name> <label-key>=<label-value>
```

这里，我们为 kubernetes-minion1 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 kubernetes-minion1 zone=north
NAME           LABELS
kubernetes-minion1   kubernetes.io/hostname=kubernetes-minion1,zone=north
STATUS
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。

然后，在 Pod 的配置文件中加入 nodeSelector 定义，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 上去。

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   kubernetes-minion1
```

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。

这种基于 Node 标签的调度方式灵活性很高，比如我们可以把一组 Node 分别贴上“开发环境”“测试验证环境”“用户验收环境”这三组标签中的一种，此时一个 Kubernetes 集群就承载了 3 个环境，这将大大提高开发效率。

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 时，即使还有其他可供调度的 Node，这个 Pod 也最终会调度失败。

4.3.6 应用的滚动升级

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 副本数量逐渐减少到 0，同时新的 RC 中的 Pod 副本数量从 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。

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
```

在配置文件中有几处需要注意：

- (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。

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-fefd9752aa5883ca4d53013a7b583967), 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
```

至此, 可以看到 Pod 恢复到更新前的版本了。

4.3.7 Kubernetes 集群高可用方案

Kubernetes 作为容器应用的管理中心, 通过对 Pod 的数量进行监控, 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 高可用性方案

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 4.10 所示，其集群部署实例如图 4.2 所示。

表 4.10 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

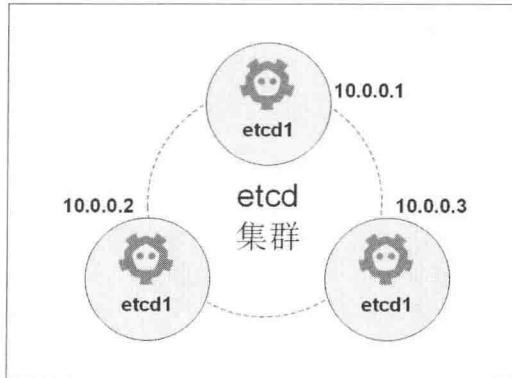


图 4.2 etcd 集群部署实例

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/etcd1"    #etcd 数据保存目录
ETCD_LISTEN_PEER_URLS= "http://10.0.0.1:2380"  #集群内部通信使用的 URL
ETCD_LISTEN_CLIENT_URLS= "http://10.0.0.1:2379"  #供外部客户端使用的 URL
.....
#[cluster]
ETCD_INITIAL_ADVERTISE_PEER_URLS= "http://10.0.0.1:2380"  #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
ETCD_ADVERTISE_CLIENT_URLS= "http://10.0.0.1:2379"  #广播给外部客户端使用的 URL
```

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/etcd2"    #etcd 数据保存目录
ETCD_LISTEN_PEER_URLS="http://10.0.0.2:2380"  #集群内部通信使用的 URL
ETCD_LISTEN_CLIENT_URLS="http://10.0.0.2:2379"  #供外部客户端使用的 URL
.....
#[cluster]
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=
etc1=http://10.0.0.1:2380,etc2=http://10.0.0.2:2380,etc3=http://10.0.0.3:2380
"      #初始集群成员列表
ETCD_INITIAL_CLUSTER_STATE="exist"        # existing 表示加入已存在的集群
ETCD_INITIAL_CLUSTER_TOKEN="etcd-cluster"  #集群名称
ETCD_ADVERTISE_CLIENT_URLS="http://10.0.0.2:2379"  #广播给外部客户端使用的 URL
```

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://10.0.0.1:
7001 clientURLs=http://10.0.0.1:2379,http://10.0.0.1:4001
acda82ba1cf790fc: name=default peerURLs=http://10.0.0.2:2380,http://10.0.0.2:
7001 clientURLs=http://10.0.0.2:2379,http://10.0.0.2:4001
eba209cd40012cd2: name=default peerURLs=http://10.0.0.3:2380,http://10.0.0.3:
7001 clientURLs=http://10.0.0.3:2379,http://10.0.0.3:4001
```

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://10.0.0.1:4001,http://10.0.0.2:4001,http://10.0.0.3:4001
```

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、NFS 网络文件系统，或者使用云服务商提供的网盘系统等来实现。

2. Kubernetes Master 组件的高可用性方案

在 Kubernetes 体系中，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，并且使用 Active-Standby-Standby 模式，保证任何时候总有一套 Master 能够正常工作。

所有工作节点上的 Kubelet 和 kube-proxy 服务则需要访问 Master 集群的统一访问入口地址，例如可以使用 pacemaker 等工具来实现。图 4.3 展示了一种典型的部署方式。

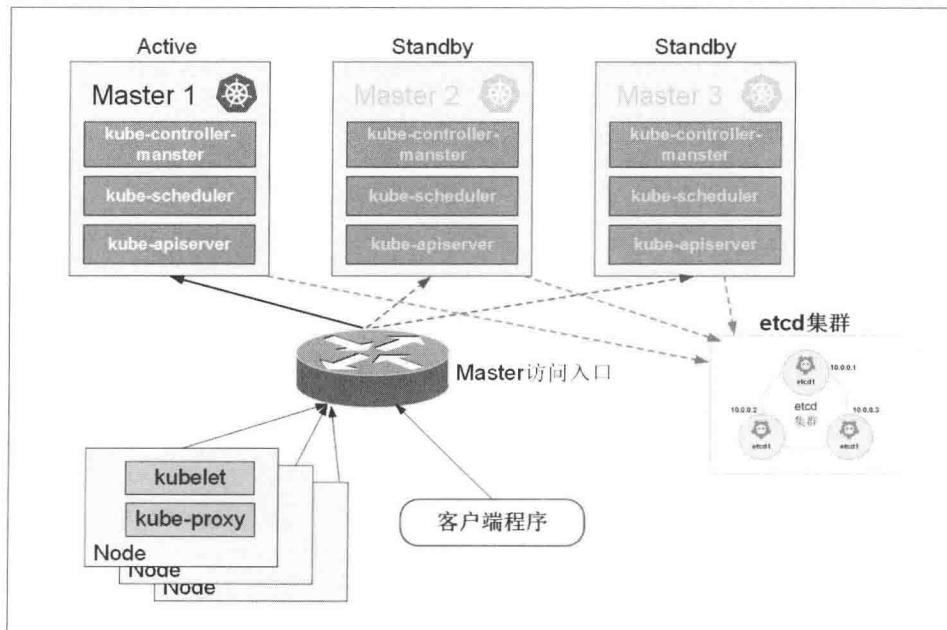


图 4.3 Kubernetes Master 高可用部署架构

4.4 资源配额管理

在第2章中介绍了资源的配额管理功能，主要用于对集群中可用的资源进行分配和限制。为了开启配额管理，我们首先要设置 kube-apiserver 的--admission_control 参数，使之加载这两个准入控制器：

```
kube-apiserver ... --admission_control=LimitRanger,ResourceQuota...
```

接下来我们将以具体案例的方式来深入学习 Kubernetes 配额管理的用法。

4.4.1 指定容器配额

对指定容器实施配额管理非常简单，只要在 Pod 或 ReplicationController 的定义文件中设定 resources 属性即可为某个容器指定配额。目前容器支持 CPU 和 Memory 两类资源的配额限制。

在下面这个 RC 定义文件中增加了 redis-master 的资源配额声明：

```
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
          resources:
            limits:
              cpu: 0.5
              memory: 128Mi
```

以上配置表示，系统将对名为 master 的容器限制 CPU 为 0.5（也可以写为 500m），可用内存限制为 128MiB 字节。

这里有必要简要说明在上述定义中所采用的 CPU 和 Memory 的单位，它们遵循国际单位制（International System of Units），包括十进制的 E、P、T、G、M、K、m，或二进制的 Ei、Pi、Ti、Gi、Mi、Ki。小于 1 的数字可以用小数来表示，此外，KiB 与 MiB 是十进制表示的字节单位，而常见的 KB 与 MB 则是二进制表示的字节单位，区别如下：

```
1 KB (kilobyte) = 1000 bytes = 8000 bits
1 KiB (kibibyte) = 2^10 bytes = 1024 bytes = 8192 bits
```

Kubernetes 启动一个容器时，会将 CPU 的配额值乘以 1024 并转为整数传递给 docker run 的--cpu-shares 参数，之所以乘以 1024 是因为 Docker 的 cpu-shares 参数是以 1024 为基数计算 CPU 时间的。另外，Docker 官方文档里解释说 cpu-shares 是一个相对权重值（Relative Weight），因此 Kubernetes 官方文档里解释以 cpu: 0.5 表示该容器占用 0.5 个 CPU 计算时间的说法其实是不准确的。仅当该节点是单核心 CPU 而且只运行两个容器，且每个容器的 CPU 配额设定为 0.5 时，上述说法才成立。假如在一个节点上同时运行了 3 个容器 A、B、C，其中 A 容器的 CPU 配额设置为 1，B 容器与 C 容器设置为 0.5，那么当系统的 CPU 利用率达到 100% 时，A 容器只占用了 $1 \times 100 / (1 + 0.5 + 0.5) = 50\%$ 的 CPU 时间，而 B 容器与 C 容器分别占用了 25% 的 CPU 时间。如果此时我们加入一个新的容器 D，它的 CPU 配额也设置为 1，则通过计算我们得到 A 容器此时只占据了 33% 的 CPU 时间。对于目前主流的多核 CPU，容器的 CPU 配额会在多核心上进行承担。因此在多核 CPU 上，即使某个容器声明 CPU<1，它也可能会占满多个 CPU 核。例如两个设定为 cpu=0.5 的容器运行在 4 核的 CPU 上，则每个容器可能会用光 $4 \times 0.5 / (0.5 + 0.5) = 2$ 个 CPU 核。

同样，Memory 配额也会被转换为整数传递给 docker run 的--memory 参数。如果一个容器在运行过程中超出了指定的内存配额，则它可能会被“杀掉”，然后重新启动。因此对容器的内存配额需要进行准确的测试和评估。CPU 配额则不会因为偶然超标使用而导致容器被系统“杀掉”。

由于 CPU 和 Memory 的限额最终涉及 Linux 的底层 cgroup 的相关知识，所以有兴趣的读者可以继续深入研究这部分的相关知识。

4.4.2 全局默认配额

除了可以直接在容器（或 RC）的定义文件中给指定的容器增加资源配额参数，我们还可以通过创建 LimitRange 对象来定义一个全局默认配额模板。这个默认配额模板会加载到集群中的每个 Pod 及容器上，这样就不用我们手工为每个 Pod 和容器重复设置了。

LimitRange 对象可以同时在 Pod 和 Container 两个级别上进行对资源配置的设置。当 LimitRange

创建生效后，之后创建的 Pod 都将使用 LimitRange 设置的资源配置进行约束。

首先，我们定义一个名为 limit-range-1 的 LimitRange，配置文件名为 pod-container-limits.yaml，下面是该文件的完整内容：

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-range-1
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "2"
        memory: 1Gi
      min:
        cpu: 250m
        memory: 32Mi
    - type: "Container"
      max:
        cpu: "2"
        memory: 1Gi
      min:
        cpu: 250m
        memory: 32Mi
      default:
        cpu: 250m
        memory: 64Mi
```

上述设置表明：

- ① 任意 Pod 内所有容器的 CPU 使用限制在 0.25~2；
- ② 任意 Pod 内所有容器的内存使用限制在 32Mi~1Gi；
- ③ 任意容器的 CPU 使用限制在 0.25~2，默认值为 0.25；
- ④ 任意容器的内存使用限制在 32Mi~1Gi，默认值为 64Mi。

接下来，使用 kubectl create 提交上述定义文件到 Kubernetes 集群里以生效：

```
$ kubectl replace -f pod-container-limits.yaml
limitranges/limit-range-1
$ kubectl describe limits limit-range-1
Name:          limit-range-1
Namespace:     default
Type          Resource      Min      Max      Default
----          -----      ---      ---      ---
Pod           memory       32Mi    1Gi     -
```

Pod	cpu	250m	2	-
Container	cpu	250m	2	250m
Container	memory	32Mi	1Gi	64Mi

最后，我们来检验上述全局配额是否起作用。

定义一个名为 redis-master-pod 的 Pod，不指定资源配额，对应的文件名为 redis-master-pod.yaml，下面是其完整定义：

```
apiVersion: v1
kind: Pod
metadata:
  name: redis-master-pod
  labels:
    name: redis-master-pod
spec:
  containers:
  - name: master-pod
    image: kubeguide/redis-master
    ports:
    - containerPort: 6379
```

运行 kubectl create 命令创建该 Pod：

```
$ kubectl create -f redis-master-pod.yaml
pods/redis-master-pod
```

创建成功后，查看该 Pod 的详细信息，可以看到系统中 LimitRange 的设置对新创建的容器进行了资源限制（使用了 LimitRange 中的默认值）：

```
$ kubectl describe pod redis-master-pod
.....
Name:           redis-master-pod
Containers:
  master-pod:
    Image:      kubeguide/redis-master
    Limits:
      memory:   64Mi
      cpu:       250m
    State:      Running
.....
```

此外，如果我们在 Pod 的定义文件中指定了配额参数，则可遵循局部覆盖全局的原则，此配额参数会“覆盖”全局参数的值。当然，如果用户指定的配额参数超过了全局设定的最大值，则会被“禁止”。在下面的例子中，我们将内存配额改为 1.5Gi，系统将创建失败：

```
$ kubectl create -f redis-master-pod.yaml
Error from server: error when creating "redis-master-pod.yaml": Pod "redis-master-pod" is forbidden: Maximum memory usage per pod is 1Gi
```

最后需要说明的一点是：LimitRange 是跟 Namespace 捆绑的，每个 Namespace 都可以关联一个不同的 LimitRange 作为其全局默认配额配置。另外，创建 LimitRange 时可以在命令行以指定--namespace=yournamespace 的方式关联到指定的 Namespace 上，也可以在定义文件中直接指定 namespace，如下面的例子：

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-range-1
  namespace: development
spec:
  limits:
    - type: "Container"
      default:
        cpu: 250m
        memory: 64Mi
```

4.4.3 多租户配额管理

多租户在 Kubernetes 中以 Namespace 来体现，这里的多租户可以是多个用户、多个业务系统或者相互隔离的多种作业环境。一个集群中的资源总是有限的，当这个集群被多个租户的应用同时使用时，为了更好地使用这种有限的共有资源，我们需要将资源配额的管理单元提升到租户级别，只需要在不同租户对应的 Namespace 上加载对应的 ResourceQuota 配置即可达到目的。

下面我们举例说明如何使用 ResourceQuota 来实现基于租户的配额管理，场景如下。

集群拥有的总资源为：CPU 共有 128core；内存总量为 1024GiB；有两个租户，分别是开发组和测试组，开发组的资源配置为 32 core CPU 及 256GiB 内存，测试组的资源配置为 96 core CPU 及 768GiB 内存。

首先，创建开发组对应的命名空间：

namespace-development.yaml:

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

接着，创建用于限定开发组的 ResourceQuota 对象，注意 metadata.namespace 属性被设定为开发组的命名空间：

resourcequota-development.yaml

```
apiVersion: v1
```

```
kind: ResourceQuota
metadata:
  name: quota-development
  namespace: development
spec:
  hard:
    cpu: "32"
    memory: 256Gi
    persistentvolumeclaims: "10"
    pods: "100"
    replicationcontrollers: "50"
    resourcequotas: "1"
    secrets: "20"
    services: "50"
```

查看 ResourceQuota 的详细信息：

```
$ kubectl describe quota quota-development --namespace=development
Name:           quota-development
Namespace:      development
Resource        Used   Hard
-----  -----  -----
cpu          0     32
memory       0     256Gi
persistentvolumeclaims 0     10
pods         0     100
replicationcontrollers 0     50
resourcequotas   1     1
secrets        0     20
services       0     50
```

重复上述步骤，为测试组也创建相应的 namespace 与 ResourceQuota，这里省略具体操作。

在创建完 ResourceQuota 之后，对于所有需要创建的 Pod 都必须指定具体的资源配额设置。

否则，创建 Pod 会失败：

```
$ kubectl create -f redis-master.yaml
Error from server: error when creating "redis-master.yaml": Pod "redis-master" is forbidden: Limited to 256Gi memory, but pod has no specified memory limit
```

可以使用前面介绍的两种办法来为 Pod 声明配额，这里省略具体操作。

在创建了一些 Pod 以后，可以通过命令 kubectl describe resourcequota 来查看某个租户的配额使用情况。

下面是对 development 租户的配额使用情况的统计：

```
$ kubectl describe resourcequota quota-development --namespace=development
Name:           quota-development
Namespace:      development
```

Resource	Used	Hard
cpu	250m	32
memory	67108864	256
Gipersistentvolumeclaims	0	10
pods	1	100
replicationcontrollers	0	50
resourcequotas	1	1
secrets	0	20
services	0	50

此外，还可以通过 `kubectl describe namespace` 命令查看一个 Namespace 内所包括的 ResourceQuota 和 LimitRange 的信息。此命令有助于帮助我们更好地去了解某个租户的配额定义和使用情况：

```
$ kubectl describe namespace development
Name: development
Labels: <none>
Status: Active

Resource Quotas
Resource          Used   Hard
---              ---   ---
cpu               0      32
memory           0      256Gi
persistentvolumeclaims 0      10
pods              0      100
replicationcontrollers 0      50
resourcequotas    1      1
secrets           0      20
services          0      50

Resource Limits
Type       Resource     Min   Max   Default
---       -----
Container  cpu          -     -     250m
Container  memory       -     -     64Mi
```

4.5 Kubernetes 网络配置方案详解

根据第 2 章对 Kubernetes 网络机制的介绍，为了实现各 Node 上 Pod 之间的互联互通，需要一些方案来打通网络，这是 Kubernetes 集群能够正常工作的前提。本节将对常用的直接路由、Flannel 和 Open vSwitch 三种配置进行详细说明。

4.5.1 直接路由方案

通过在每个 Node 上添加到其他 Node 上 docker0 的静态路由规则，就可以将不同物理服务器上 Docker Daemon 创建的 docker0 网桥互通。图 4.4 描述了在两个 Node 之间打通网络的情况。

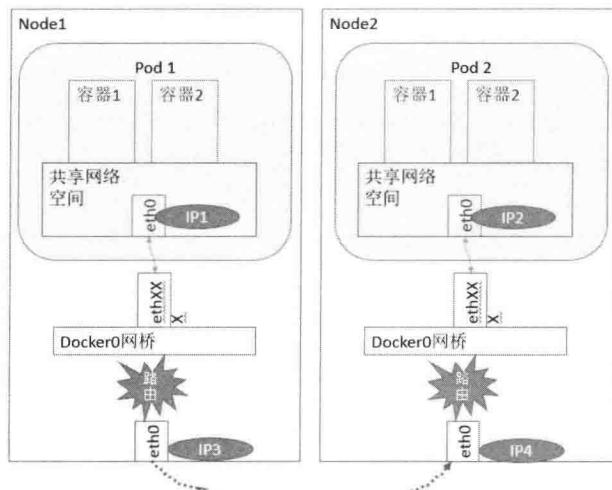


图 4.4 直接路由方式实现 Pod 到 Pod 的通信

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

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>。

除了在每台服务器安装 Quagga 软件并启动，还可以使用互联网上的一个 Quagga 容器来运行，在本例中使用 index.alauda.cn/georce/router 镜像启动 Quagga。在每台 Node 上下载该 Docker 镜像：

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

在运行 Quagga 路由器之前，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，使用 ifconfig 命令修改 docker0 网桥的地址和子网（假设 Node 所在的物理网络不是 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。

4.5.2 使用 flannel 叠加网络

flannel 采用叠加网络（Overlay Network）模型来完成网络的打通，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。

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-opts.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
EnvironmentFile=-/etc/sysconfig/docker-network
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://192.168.1.128:4001"

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

# Any additional options that you want to pass
#FLANNEL_OPTIONS= ""
```

在启动 flannel 之前，需要在 etcd 中添加一条网络配置记录，这个配置将用于 flannel 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) 在每个 Node 节点执行以下命令来完成对 docker0 网桥的设置：

```
# 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
```

这样即可完成 flannel 叠加网络的设置。

使用 ping 命令验证各 Node 上 docker0 之间的相互访问。在 10.1.10.1 机器上 ping 10.1.30.1 (另一台机器)：

```
$ 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
....
```

在 etcd 中也可以查看到 flannel 设置的 flannel0 地址与物理机 IP 地址的路由规则：

```
# 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" }
```

4.5.3 使用 Open vSwitch

以两个 Node 为例，目标网络拓扑如图 4.5 所示。

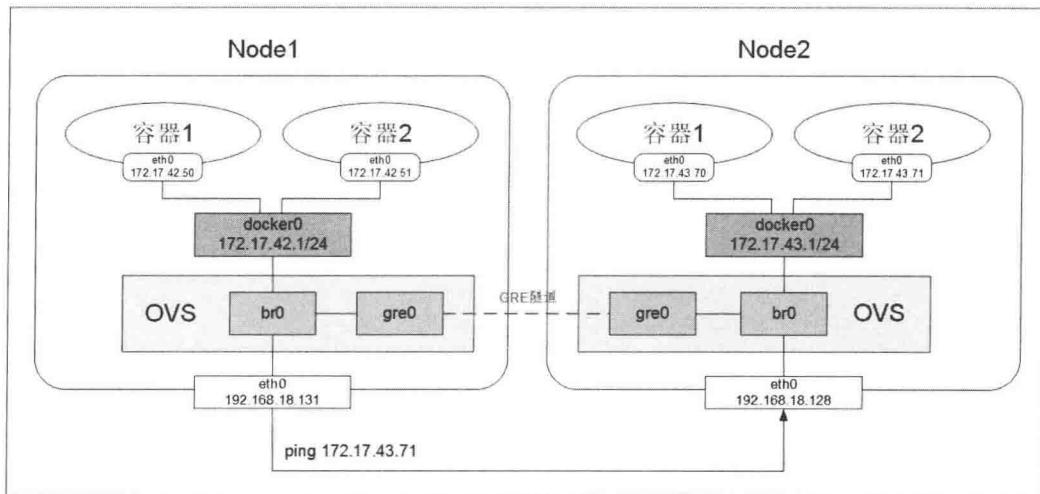


图 4.5 通过 Open vSwitch 打通网络

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 DockerDaemon 进程里的 bip 参数。

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 功能，配置后重启机器：

```
# 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：

```
[root@docker131 ~]# 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
    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：

```
[root@docker128 ~]# 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
    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：

```
[root@docker128 ~]# 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：

```
[root@docker128 ~]# 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：

```
[root@docker128 ~]# 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)
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)
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组成一个网状网，才能实现全网互通。

4.6 Kubernetes 集群监控

4.6.1 使用 kube-ui 查看集群运行状态

Kubernetes 自带一个集群状态的 Web 图形化显示界面，便于运维人员查看当前集群的运行状态。

首先需要启动 kube-ui 服务和 Pod。

在 Kubernetes 的安装包 kubernetes.tar.gz 中的 cluster/addons/kube-ui 目录下，有 kube-ui-rc.yaml 和 kube-ui-svc.yaml 文件。

kube-ui-rc.yaml 的内容为：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-ui-v1
  namespace: kube-system
  labels:
    k8s-app: kube-ui
    version: v1
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-ui
    version: v1
  template:
    metadata:
      labels:
        k8s-app: kube-ui
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      containers:
      - name: kube-ui
        image: gcr.io/google_containers/kube-ui:v1.1
        resources:
          limits:
            cpu: 100m
            memory: 50Mi
        ports:
        - containerPort: 8080
```

kube-ui-svc.yaml 的内容为：

```
apiVersion: v1
kind: Service
metadata:
  name: kube-ui
  namespace: kube-system
  labels:
    k8s-app: kube-ui
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: "KubeUI"
spec:
  selector:
    k8s-app: kube-ui
  ports:
  - port: 80
    targetPort: 8080
```

通过 kubectl create 命令完成创建：

```
$ kubectl create -f kube-ui-rc.yaml
$ kubectl create -f kube-ui-svc.yaml
```

启动成功后，通过 Kubernetes Master 的 IP 地址来访问 kube-ui：

<https://<kubernetes-master>:<port>/ui>

该 URL 将会被重定向到：

<https://<kubernetes-master>:<port>/api/v1/proxy/namespaces/kube-system/services/kube-ui/#/dashboard>

图 4.6 显示了 kube-ui 的主页，展示了所有 Node 的信息，并且每秒刷新显示每个 Node 的 CPU 使用率、内存使用情况和文件系统的使用情况。

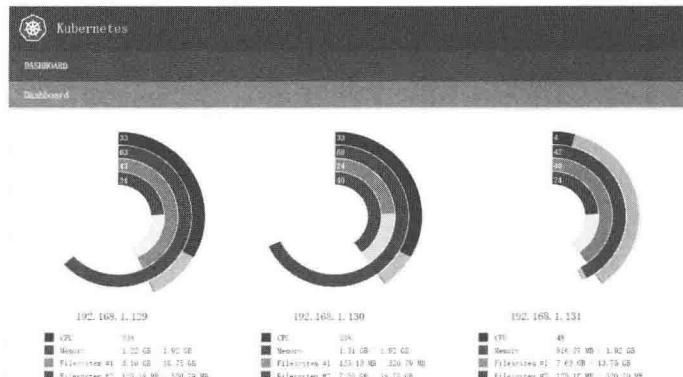


图 4.6 kube-ui 主页

单击右侧按钮“Views”，可以选择查看 Explore、Pods、Nodes、Replication Controllers、Services 和 Events 等信息。

其中，Explore 选项可以在一页内显示当前集群中用户创建的 Pod、ReplicationController 和 Service，如图 4.7 所示。

The screenshot shows the Kubernetes Explore page. At the top, there's a navigation bar with 'Dashboard' and 'Explore'. Below it, a 'Group by' dropdown is set to 'type'. Under 'Type: pod', there are six entries: 'frontend-9ubp3' (status: green), 'frontend-balog8' (status: green), 'frontend-zxygo' (status: green), 'redis-master-neul0' (status: green), 'redis-slave-5hadv' (status: green), and 'redis-slave-foxfu' (status: green). Under 'Type: replicationController', there are three entries: 'frontend' (status: green), 'redis-master' (status: green), and 'redis-slave' (status: green). Under 'Type: service', there are five entries: 'cassandra' (status: green), 'frontend' (status: green), 'kubernetes' (status: green), 'nibus' (status: green), 'redis-master' (status: green), 'redis-slave' (status: green), and 'zookeeper' (status: green).

图 4.7 Explore 页面

单击左上方的 Group By 下拉列表，可以使用 Type、Name、host、component 或 provider 作为分组条件进行分组显示，如图 4.8 所示。

This screenshot shows the same Explore page as Figure 4.7, but with a different 'Group by' setting. The 'Group by' dropdown now has 'host' selected. The resource list remains the same, but the grouping is now based on the host they are running on. The hosts listed are 'md-balog8', 'frontend-zxygo', 'redis-master-neul0', 'redis-slave-5hadv', and 'redis-slave-foxfu'.

图 4.8 Group By 条件

单击每个资源对象右侧的小三角符号，可以选择 FILTER 来过滤并显示满足条件的结果，如图 4.9 所示。



图 4.9 资源的 FILTER 过滤条件

直接单击某个资源实例的链接，即可查看其详细信息，如图 4.10 所示。

This screenshot shows the detailed view of a Pod named 'frontend-xzqyo'. The top navigation bar includes 'Dashboard' and 'Pod'. Below the title, there is a 'BACK' link. The main content area displays the following details for the Pod:

- Name:** frontend-xzqyo
- Status:** Running on 192.168.1.130
- Created:** Sep 9, 2016 5:38:54 PM
- Host Networking:** 172.168.1.130/192.168.1.130
- Pod Networking:** 10.11.20.4:480
- Labels:** runer=frontend
- Containers:**

Name	Image	Daily	Restarting	State
nginx-redis	192.168.1.128:1480/kubernetes/example-guestbook-nginx-redis	true	0	Running
				Started: Sep 10, 2016 3:29:42 AM

图 4.10 Pod 的详细信息

单击右侧按钮 Views 的其他选项，即可查看相应的资源状态，例如查看全部 Pod 的页面，如图 4.11 所示。

The screenshot shows the Kubernetes Dashboard interface. At the top, there's a navigation bar with the Kubernetes logo and the word 'Kubernetes'. Below it is a secondary navigation bar with 'Dashboard' and 'Pods'. On the right side of this bar is a 'Views' dropdown menu. The main area is titled 'PODS' and contains a table listing several pods. The columns in the table are: Pod, IP, Status, Container(s), Image(s), Port, and Labels. The table lists the following pods:

Pod	IP	Status	Container(s)	Image(s)	Port	Labels
frontend-9ubp2	10.1.30.12	Running	php-redis	192.168.1.128:1180/kubernetes/example-guestbook-php-redis	102.168.1.101	name: frontend
frontend-ba0q8	10.1.10.6	Running	php-redis	192.168.1.128:1180/kubernetes/example-guestbook-php-redis	102.168.1.129	name: frontend
frontend-xzqyo	10.1.20.6	Running	php-redis	192.168.1.128:1180/kubernetes/example-guestbook-php-redis	102.168.1.130	name: frontend
redis-master-neu00	10.1.30.5	Running	master	192.168.1.128:1180/kubernetes/redis-master	102.168.1.130	name: redis-masters
redis-slave-Shdw	10.1.10.4	Running	slave	192.168.1.128:1180/kubernetes/redis-slave:2.0	102.168.1.129	name: redis-slave
redis-slave-foxuf	10.1.20.4	Running	slave	192.168.1.128:1180/kubernetes/redis-slave:2.0	102.168.1.130	name: redis-slave-2.0

At the bottom left of the table, there's a 'Rows count per page' dropdown set to 50.

图 4.11 查看全部 Pod 的页面

4.6.2 使用 cAdvisor 查看容器运行状态

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 定义了 cAdvisor 对外提供服务的口号，默认为 4194。

可以通过浏览器访问 cAdvisor 提供的 Web 页面。假设 Kubernetes 集群中的一个 Node 的 IP 地址是 192.168.1.129，则在浏览器中输入网址 <http://192.168.1.129:4194> 来访问 cAdvisor 的监控页面。cAdvisor 的主页显示了主机实时运行状态，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 4.12 展示了 cAdvisor 的几个性能监控页面。

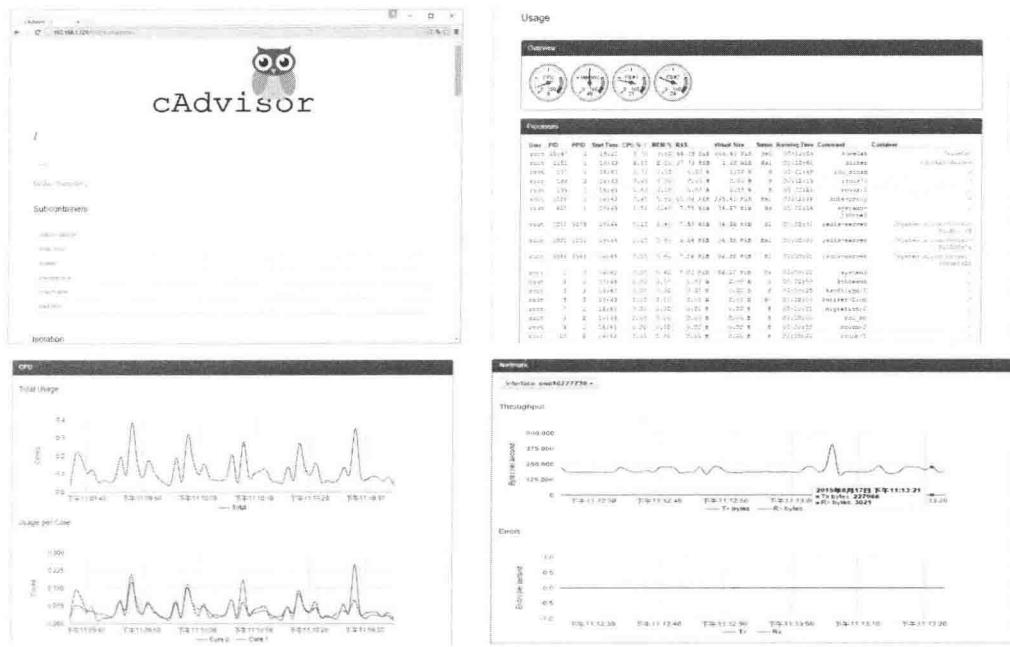


图 4.12 主机的性能监控页面

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 4.13 所示。

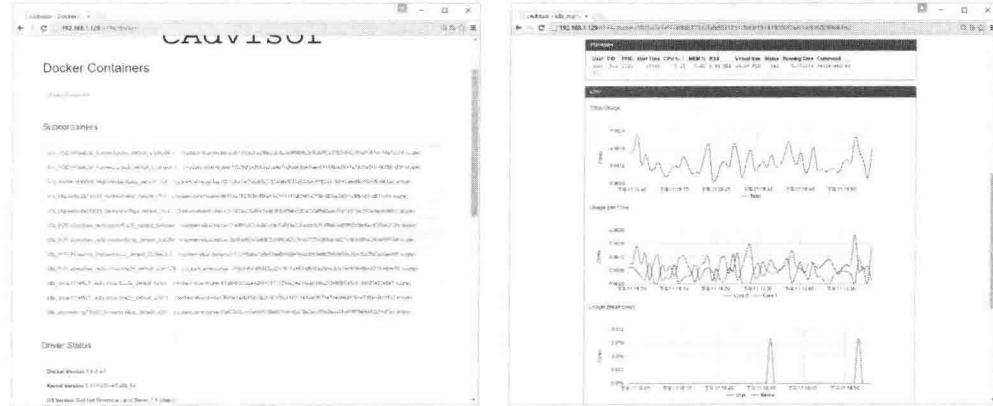


图 4.13 容器的性能监控页面

此外，cAdvisor 也提供 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://192.168.1.129:4194/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/sda1",  
            "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": [  
                {  
                    "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": 4100000000,
                    "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。它们的安装和使用说明参见第 5 章。

4.7 Trouble Shooting 指导

如果 Kubernetes 在运行中出现某些故障，则我们可以通过多种手段来跟踪和发现问题，流程如下。

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。

4.7.1 对象的 Event 事件

在 Kubernetes 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:          kubernetes-minion1/192.168.1.129
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 kubernetes-minion1}  implicitly required container POD pulled
Pod container image "192.168.1.128:1180/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 kubernetes-minion1}  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 kubernetes-minion1}    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 kubernetes-minion1}    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 kubernetes-minion1}    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 kubernetes-minion1
```

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；
- ◎ 开启了资源配额管理并且当前 Pod 的目标节点上恰好没有可用的资源。

`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:            10.254.208.57
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
```

4.7.2 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。

4.7.3 Kubernetes 系统日志

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)
      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 的标准输出，则也可以通过另外一些服务的启动参数来指定日志的存放目录。

- ① --logtostderr=false：不输出到 stderr。
- ② --log-dir=/var/log/kubernetes：日志存放目录。
- ③ --alsologtostderr=false：设置为 true 则表示将日志输出到文件时也输出到 stderr；
- ④ --v=0：glog 日志级别；
- ⑤ --vmodule=gfs*=2,test*=4：glog 基于模块的详细日志级别；

在--log_dir 设置的目录中可以看到每个进程生成了一些日志文件，日志文件的数量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为：

- ① kube-controller-manager.ERROR。
- ② kube-controller-manager.INFO。
- ③ kube-controller-manager.WARNING。
- ④ kube-controller-manager.kubernetes-master.unknownuser.log.ERROR.20150930-173939.9847。
- ⑤ kube-controller-manager.kubernetes-master.unknownuser.log.INFO.20150930-173939.9847。
- ⑥ 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，但可能会遇到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。

4.7.4 常见问题

本节对 Kubernetes 系统中的几个常见问题及解决方法进行说明。

1. Pod 一直处于 Pending 的状态，无法完成镜像的下载

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: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)
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:0.8.0": 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:0.8.0，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/pause:latest
```

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 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，并且是成功结束的 (ExitCode=0)。然后，根据 RC 的定义，为了保持 Pod 副本的数量，Kubernetes 再次创建并启动了一个新的容器。

新的容器仍然会很快结束，然后 Kubernetes 会再次创建另一个新的容器，进入一个无限往复的过程中。

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
```

4.7.5 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。

第 5 章

Kubernetes 高级案例进阶

本章将通过几个复杂的案例对 Kubernetes 进行实战操作，以进一步加深对 Kubernetes 系统核心概念和工作机制的理解。

5.1 Kubernetes DNS 服务配置案例

在 Kubernetes 系统中，Pod 在访问其他 Pod 的 Service 时，可以通过两种服务发现方式完成，即环境变量和 DNS 方式。但是使用环境变量是有限制条件的，即 Service 必须在 Pod 之前被创建出来，然后系统才能在新建的 Pod 中自动设置与 Service 相关的环境变量。DNS 则没有这个限制，其通过提供全局的 DNS 服务器来完成服务的注册与发现。

Kubernetes 提供的 DNS 由以下三个组件组成。

- (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。

这三个组件以 Pod 的方式启动和运行，所以在一个 Kubernetes 集群中，它们都可能被调度到任意一个 Node 节点上去。为了能够使它们之间网络互通，需要将各 Pod 之间的网络打通，如何打通网络请参考第 4 章网络配置部分的详细说明。图 5.1 描述了 Kubernetes DNS 服务的整体架构。

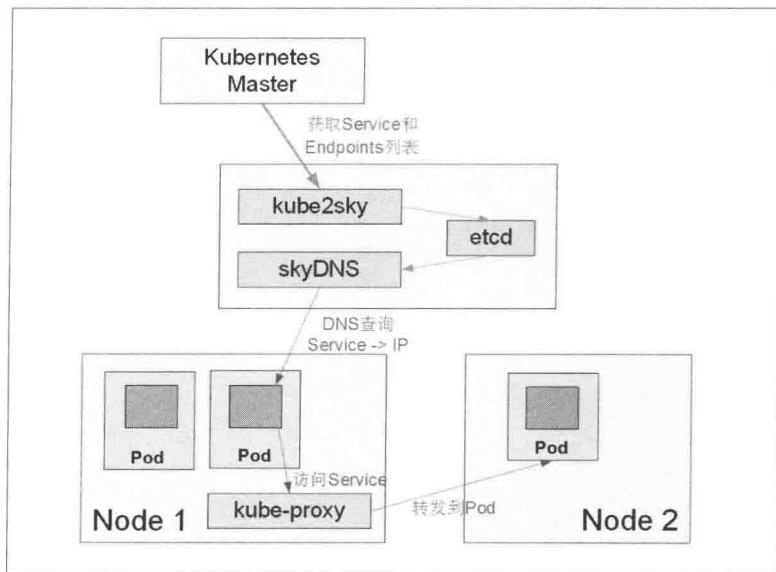


图 5.1 Kubernetes DNS 总体架构

网络配置完成后，通过创建 RC 和 Service 来启动 DNS 服务。

5.1.1 skydns 配置文件

首先创建 DNS 服务的 ReplicationController 配置文件 skydns-rc.yaml，在这个 RC 配置中包含了 3 个 Container 的定义：

```

apiVersion: v1
kind: ReplicationController
metadata:
  name: kube-dns-v8
  namespace: kube-system
  labels:
    k8s-app: kube-dns
    version: v8
    kubernetes.io/cluster-service: "true"
spec:
  replicas: 1
  selector:
    k8s-app: kube-dns
    version: v8
  template:
    metadata:

```

```
labels:
  k8s-app: kube-dns
  version: v8
  kubernetes.io/cluster-service: "true"
spec:
  containers:
  - name: etcd
    image: gcr.io/google_containers/etcd:2.0.9
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
    command:
      - /usr/local/bin/etcd
      - --data-dir
      - /var/etcd/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: /var/etcd/data
  - name: kube2sky
    image: gcr.io/google_containers/kube2sky:1.11
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
    args:
      # command = "/kube2sky"
      - --kube_master_url=http://192.168.1.128:8080
      - --domain=cluster.local
  - name: skydns
    image: gcr.io/google_containers/skydns:2015-03-11-001
    resources:
      limits:
        cpu: 100m
        memory: 50Mi
    args:
      # command = "/skydns"
      - --machines=http://localhost:4001
      - --addr=0.0.0.0:53
      - --domain=cluster.local
  ports:
```

```

    - containerPort: 53
      name: dns
      protocol: UDP
    - containerPort: 53
      name: dns-tcp
      protocol: TCP
  volumes:
    - name: etcd-storage
      emptyDir: {}
  dnsPolicy: Default

```

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://192.168.1.128:8080。

(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 会监听 Kubernetes，当有新的 Service 创建时，就会生成相应的记录并保存到 etcd 中。kube2sky 为每个 Service 生成两条记录：

- ❶ <service_name>.<namespace_name>.<domain>;
- ❷ <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。

创建 DNS 服务的 Service 配置文件如下：

```

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: 20.1.0.100
  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。

5.1.2 修改每个 Node 上的 Kubelet 启动参数

修改每台 Node 上 Kubelet 的启动参数：

- ◎ --cluster_dns=20.1.0.100，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。

5.1.3 创建 skydns Pod 和服务

之后，通过 kubectl create 完成 RC 和 Service 的创建：

```
# kubectl create -f skydns-rc.yaml  
# kubectl create -f skydns-svc.yaml
```

创建完成后，查看到系统创建的 RC、Pod 和 Service 都已创建成功：

```
# kubectl get rc --namespace=kube-system  
CONTROLLER      CONTAINER(S)        IMAGE(S)          SELECTOR          REPLICAS  
kube-dns-v8     etcd              kubeguide/etcd:2.0.9  
k8s-app=kube-dns,version=v8   1  
                  kube2sky         kubeguide/kube2sky:1.11  
                  skydns          kubeguide/skydns:2015-03-11-001  
  
# kubectl get pods --namespace=kube-system  
NAME            READY   STATUS    RESTARTS   AGE  
kube-dns-v8-0r71x   3/3    Running   0          24m  
  
# kubectl get services --namespace=kube-system  
NAME           LABELS          SELECTOR          IP(S)          PORT(S)  
kube-dns  
  
k8s-app=kube-dns,kubernetes.io/cluster-service=true,kubernetes.io/name=KubeDNS  
k8s-app=kube-dns   20.1.0.100   53/UDP  
                                53/TCP
```

然后，我们创建一个普通的 Service，以 redis-master 服务为例：

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
```

查看创建出来的 Service:

```
# kubectl get services
```

NAME	LABELS	SELECTOR	IP(S)	PORT(S)
redis-master	name=redis-master	name=redis-master	20.1.231.244	6379/TCP

可以看到，系统为 redis-master 服务分配了一个 IP 地址：20.1.231.244。

5.1.4 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:
  - image: gcr.io/google_containers/busybox
    command:
    - sleep
    - "3600"
  imagePullPolicy: IfNotPresent
  name: busybox
  restartPolicy: Always
```

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: 20.1.0.100
Address 1: 20.1.0.100

Name: redis-master
Address 1: 20.1.231.244
```

可以看到，通过 DNS 服务器 20.1.0.100 成功找到了名为“redis-master”服务的 IP 地址：20.1.231.244。

如果某个 Service 属于自定义的命名空间，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: 20.1.0.100
Address 1: 20.1.0.100 ignencfch3-v804.csc.com

Name: kube-dns.kube-system
Address 1: 20.1.0.100 ignencfch3-v804.csc.com
```

如果仅使用“kube-dns”来进行查找，则将会失败：

```
nslookup: can't resolve 'kube-dns'
```

5.1.5 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": "20.1.231.244", "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 20.1.231.244。

(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 20.1.0.100
search default.svc.cluster.local svc.cluster.local cluster.local localdomain
```

通过名字服务器 20.1.0.100 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。

例如，设置 redis-slave 的启动脚本为：

```
redis-server --slaveof redis-master 6379
```

创建 redis-slave 的 Pod 并启动它。

之后，我们可以登录 redis-slave 容器中查看，其通过 DNS 域名服务找到了 redis-master 的 IP 地址 20.1.231.244，并成功建立了连接。

通过 DNS 设置，对于其他 Service（服务）的查询将可以不再依赖系统为每个 Pod 创建的环境变量，而是直接使用 Service 的名字就能对其进行访问，使得应用程序中的代码更简洁了。

5.2 Kubernetes 集群性能监控案例

在 Kubernetes 系统中，使用 cAdvisor 对 Node 所在主机资源和在该 Node 上运行的容器进行监控和性能数据采样（详见第 4 章的描述）。由于 cAdvisor 集成在 Kubelet 中，即运行在每个 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，我们需要对所有 Node 和全部容器进行性能监控，Kubernetes 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：是对集群中各 Node、Pod 的资源使用数据进行采集的系统，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。之后 Heapster 进行数据聚合，并将结果保存到后端存储系统中。Heapster 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>。

- ③ 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 REST API 用于数据的存储和查询。InfluxDB 的主页为 <http://InfluxDB.com>。
- ④ 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>。

总体架构如图 5.2 所示。

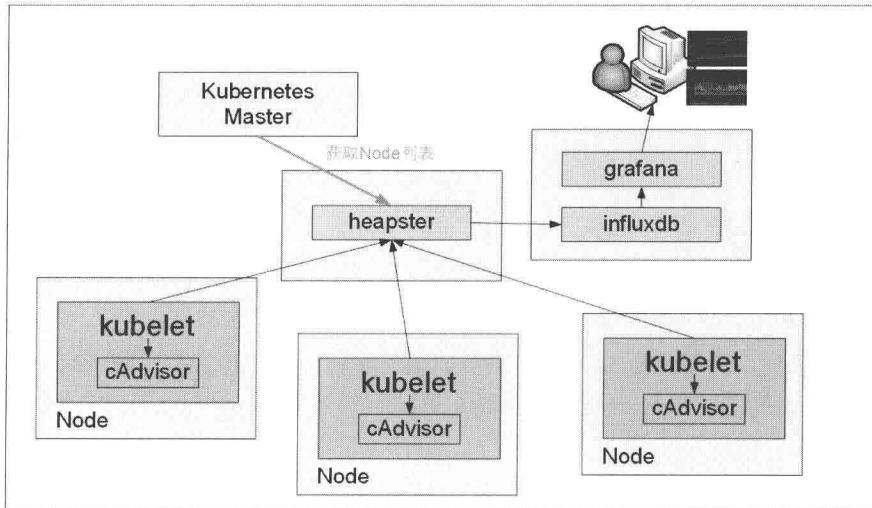


图 5.2 Heapster 集群监控系统架构图

在 Kubernetes 的当前版本中，Heapster、InfluxDB 和 Grafana 均以 Pod 的形式启动和运行。在启动这些 Pod 之前，首先需要为 Heapster 配置与 Master 的安全连接。

5.2.1 配置 Kubernetes 集群的 ServiceAccount 和 Secret

Heapster 当前版本需要使用 HTTPS 的安全方式与 Kubernetes Master 进行连接，所以需要先进行 ServiceAccount 和 Secret 的创建。如果不使用 Secret，则 Heapster 启动时将会报错：

```
/var/run/secret/kubernetes.io/serviceaccount/token no such file or directory
```

然后 Heapster 容器会被 ReplicationController 反复销毁、创建，无法正常工作。

关于 ServiceAccount 和 Secret 的原理详见第 2 章的说明。

在进行以下操作时，我们假设在 Kubernetes 集群中没有创建过 Secret（如果之前创建过，则可以先删除 etcd 中与 Secret 相关的键值）。

首先，使用 OpenSSL 工具在 Master 服务器上创建一些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
# openssl req -new -key server.key -subj "/CN=kubernetes-master" -out server.csr
# openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server.crt -days 5000
```

注意，在生成 server.csr 时-subj 参数中/CN 指定的名字需为 Master 的主机名。另外，在生成 ca.crt 时-subj 参数中/CN 的名字最好与主机名不同，设为相同可能导致对普通 Master 的 HTTPS 访问认证失败。

执行完成后会生成 6 个文件：ca.crt、ca.key、ca.srl、server.crt、server.csr、server.key。

将这些文件复制到/var/run/kubernetes/目录中，然后设置 kube-apiserver 的启动参数：

```
--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
```

之后重启 kube-apiserver 服务。

接下来，给 kube-controller-manager 服务添加以下启动参数：

```
--service_account_private_key_file=/var/run/kubernetes/apiserver.key
--root-ca-file=/var/run/kubernetes/ca.crt
```

然后重启 kube-controller-manager 服务。

在 kube-apiserver 服务成功启动后，系统会自动为每个命名空间创建一个 ServiceAccount 和一个 Secret（包含一个 ca.crt 和一个 token）：

```
# kubectl get serviceaccounts --all-namespaces
NAMESPACE     NAME      SECRETS
default       default    1
kube-system   default    1

# kubectl get secrets --all-namespaces
NAMESPACE     NAME           TYPE          DATA
default       default-token-lhx52  kubernetes.io/service-account-token  2
kube-system   default-token-23f6f  kubernetes.io/service-account-token  2

# kubectl describe secret default-token-lhx52
Name:         default-token-lhx52
Namespace:    default
Labels:       <none>
Annotations:
```

```
kubernetes.io/service-account.name=default,kubernetes.io/service-account.uid=6e0
9f5b5-52d0-11e5-a4f1-000c2921ba71
  Type:  kubernetes.io/service-account-token
  Data
  ====
  ca.crt: 1099 bytes
  token:  eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJrdWJlc5ldGVzL3Nlcn
ZpY2VhY2NvdW50Iiwia3ViZXJuZXRLcy5pbv9zZXJ2aWN1YWNjb3VudC9uYW1lc3BhY2UiOijkZWZhdw
x0Iiwia3ViZXJuZXRLcy5pbv9zZXJ2aWN1YWNjb3VudC9zZWNyZXQubmFtZSI6ImRlZmFlbHQtdG9rZW
4tbGh4NTIiLCJrdWJlc5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIj
oiZGVmYXVsdCIsImt1YmVybmV0ZXMuaw8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZC
I6IjZ1MDlmNW11LTUyZDAtMTF1NS1hNGYxLTaWMGMyOTIxYmE3MSIsInN1YiI6InN5c3R1bTpzzXJ2aW
N1YWNjb3VudDpkZWZhdWx0OmRlZmFlbHQifQ.Qln9LHiB807ztZ3ZKb1XJT3VgGCDSTMiY1uGr3QUaxh
5gGbW4EakrR_fFDiecrYQMCzXpQRkppjKutbCITD0pevcwatMVLpJXHV774xMuGmdV_tilQHETSpN-h
bSKL8CPzGpVoAurXuti3dSnwyM6K5icC9TBZKc-NYSalraFaurMaqpjBKVUbKVUbfECD5qsG8BQDjmg
1Wyg1YpnmmQjLe1DYTyMDnU3RJ1g8IYdyOizxod8-F--89pFz-_f0KYA_z3MvRnS8hdhmh5zvXik6IruG
yF-t50xRSxISg1q-idn7k3jlwmclBu03aBLSD9GvSdKhx6aESPOc65JQ

# kubectl describe secret default-token-23f6f --namespace=kube-system
Name:          default-token-23f6f
Namespace:     kube-system
Labels:        <none>
Annotations:

kubernetes.io/service-account.name=default,kubernetes.io/service-account.uid=6e0
86c37-52d0-11e5-a4f1-000c2921ba71
  Type:  kubernetes.io/service-account-token
  Data
  ====
  ca.crt: 1099 bytes
  token:  eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJrdWJlc5ldGVzL3Nlcn
ZpY2VhY2NvdW50Iiwia3ViZXJuZXRLcy5pbv9zZXJ2aWN1YWNjb3VudC9uYW1lc3BhY2UiOijkZWJ1LX
N5c3R1bSIsImt1YmVybmV0ZXMuaw8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOijkZWZhdWx0LX
Rva2VuLTizZjZmiIwia3ViZXJuZXRLcy5pbv9zZXJ2aWN1YWNjb3VudC9zZXJ2aWN1LWFjY291bnQubm
FtZSI6ImRlZmFlbHQiLCJrdWJlc5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC
51aWQioiI2ZT14NmMzNy01MmQwLTExZTUtYTRmMS0wMDBjMjkyMWJhNzEiLCJzdWIiOijzeXN0ZW06c2
VydmljZWFjY291bnQ6a3ViZS1zeXN0ZW06ZGVmYXVsdCJ9.BMeLAAGKUokPj0BD1KyUHjBjH7izM69pP
DgSiBynzOz9nsMpQyFUGK_wvloVgF0b-RVAaHkK90UfSbfzHdp6F-fc9bCbLRTF44QTmMfVbmVgwDM2k
Y1_q-EfyDE39aijJ3AscPVUNjVOb07f_Md_-htCeHWDkc7P6XpgMp4bYMUUBVfCLYLVTjujyjQLaD5EF0H
EWd8-9zjJB7_rhbPiQ-Z4N0cE3ik9cuFAshveLgqzpwdM463E_p1wyvdmKZ3EnPZrdp-2KU2AVt0Jmu
U0PlngTjf7U1Q0IndbETG1k03otbMKd3qY6Dpj2yAGyB-KNOdx8dBvaHTdzUawq2vpzg
```

之后 ReplicationController 在创建 Pod 时，会生成类型为 Secret 的 Volume 存储卷（参见第 1 章中对 Volume 的说明），并将该 Volume 挂载到 Pod 内的如下目录中：/var/run/secrets/kubernetes.io/serviceaccount。然后，容器内的应用程序就可以使用该 Secret 与 Master 建立 HTTPS 连接了。Pod 的 Volumes 设置和挂载操作由 ReplicationController 和 Kubelet 自动完成，可以通过查看 Pod 的详细信息了解到。

```
# kubectl get pods kube-dns-v8-iknnc --namespace=kube-system -o yaml
apiVersion: v1
.....
spec:
  containers:
    .....
    volumeMounts:
      - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
        name: default-token-23f6f
        readOnly: true
    .....
  serviceAccount: default
  serviceAccountName: default
  volumes:
    - name: default-token-23f6f
      secret:
        secretName: default-token-23f6f
status:
.....
```

进入容器，查看/var/run/secrets/kubernetes.io/serviceaccount 目录，可以看到两个文件 ca.crt 和 token，这两个文件就是与 Master 通信时所需的证书和秘钥信息。

5.2.2 部署 Heapster、InfluxDB、Grafana

在 ServiceAccount 和 Secrets 创建完成后，我们就可以创建 Heapster、InfluxDB 和 Grafana 等 ReplicationController 和 Service 了。

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: 30083
    - name: api
      port: 8086
      targetPort: 8086
      nodePort: 30086
  selector:
    name: influxGrafana
```

注意，这里使用 `type=NodePort` 将 InfluxDB 暴露在宿主机 Node 的端口上，以便客户端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: 30080
  selector:
    name: influxGrafana
```

同样使用 `type=NodePort` 将 Grafana 暴露在 Node 的端口上，以便客户端浏览器对其进行访问。

创建 Heapster RC：

heapster-controller.yaml

```

apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    k8s-app: heapster
    name: heapster
    version: v6
  name: heapster
  namespace: kube-system
spec:
  replicas: 1
  selector:
    name: heapster
    k8s-app: heapster
    version: v6
  template:
    metadata:
      labels:
        k8s-app: heapster
        version: v6
    spec:
      containers:
        -
          image: gcr.io/google_containers/heapster:v0.17.0
          name: heapster
          command:
            - /heapster
            - --source=kubernetes:http://192.168.1.128:8080?inClusterConfig=
false&kubeletHttps=true&useServiceAccount=true&auth=
            - --sink=InfluxDB:http://monitoring-InfluxDB:8086

```

Heapster 需要设置的启动参数如下。

1) --source

为配置监控来源。在本例中使用 kubernetes: 表示从 Kubernetes Master 获取各 Node 的信息。在 URL 后面的参数部分，修改 kubeletHttps、inClusterConfig、useServiceAccount 的值，并设置 auth 的值为空：

```
--source=kubernetes:http://192.168.1.128:8080?inClusterConfig=false&kubeletH
tts=true&useServiceAccount=true&auth=
```

URL 中可配置的参数如下。

- (1) IP 地址和端口号：为 Kubernetes Master 的地址。
- (2) kubeletPort：默认为 10255（Kubelet 服务的只读端口号）。

- (3) kubeletHttps: 是否通过 HTTPS 方式连接 Kubelet, 默认为 false。
- (4) apiVersion: API 版本号, 默认为 Kubernetes 系统的版本号, 当前为 v1。
- (5) inClusterConfig: 是否使用 Heapster 命名空间中的 ServiceAccount, 默认为 true。
- (6) insecure: 是否信任 Kubernetes 证书, 默认为 false。
- (7) auth: 客户端认证授权文件, 当 ServiceAccount 不可用时对其进行设置。
- (8) useServiceAccount: 是否使用 ServiceAccount, 默认为 false。

2) --sink

为配置后端的存储系统, 在本例中使用 InfluxDB 系统:

```
--sink=InfluxDB:http://monitoring-InfluxDB:8086
```

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。如何配置 DNS 请参见 5.1 节的案例描述。

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。因此, 使用带上命名空间的服务名也是可以的, 例如 `http://monitoring-InfluxDB.kube-system:8086`。

创建 InfluxDB 和 Grafana 的 RC 配置, 这两个容器将运行在同一个 Pod 中:

InfluxDB-Grafana-controller.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: influxGrafana
  name: influxdb-Grafana
  namespace: kube-system
spec:
  replicas: 1
  selector:
    name: influxGrafana
  template:
    metadata:
      labels:
        name: influxGrafana
    spec:
      containers:
        - image: gcr.io/google_containers/heapster_InfluxDB:v0.3
          name: InfluxDB
```

```

ports:
  - containerPort: 8083
    hostPort: 8083
  - containerPort: 8086
    hostPort: 8086
- image: gcr.io/google_containers/heapster_Grafana:v0.7
  name: Grafana
  ports:
    - containerPort: 8080
      hostPort: 8080
  env:
    - name: INFLUXDB_HOST
      value: monitoring-InfluxDB

```

注意，给 Grafana 容器定义环境变量，以供其找到 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 等本机 IP 地址也是可以访问到 InfluxDB 服务的。

最后，使用 kubectl create 命令完成所有 Service 和 RC 的创建：

```

$ kubectl create -f heapster-service.yaml
$ kubectl create -f InfluxDB-service.yaml
$ kubectl create -f Grafana-service.yaml
$ kubectl create -f InfluxDB-Grafana-controller.json
$ kubectl create -f heapster-controller.yaml

```

通过 kubectl get pods --namespace=kube-system 确认各 Pod 都成功启动了。

5.2.3 查询 InfluxDB 数据库中的数据

让我们先看一下 InfluxDB 的管理页面。

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 30083 端口访问 InfluxDB 数据库提供的管理页面，如图 5.3 所示。



图 5.3 InfluxDB 管理页面

输入默认的用户名和密码（root/root）登录后，就能对数据库进行查询了。

注意，在 Hostname 中需要填写 InfluxDB Pod 所在物理主机的 IP 地址。

如图 5.4 所示，Heapster 已经在 InfluxDB 中创建了一个名为 k8s 的数据库。

The screenshot shows the InfluxDB web interface under the 'Databases' tab. At the top, there are tabs for 'Databases', 'Cluster Admins', and 'Cluster'. The user is connected as 'root' on IP '192.168.1.130'. The main area is titled 'Databases' and lists two entries: 'grafana' and 'k8s'. Each entry has an 'Explore Data' link. Below the list, there's a section for creating a new database, with a 'Database Name' input field containing 'k8s'. Under 'Shard Spaces', there's a table with one row for 'default'. The table columns are 'Name', 'Retention Duration', 'RegEx', 'RF', and 'Split'. The 'RF' dropdown is set to '1'. A 'Remove' button is available for the default shard space. At the bottom right of the page is a large 'Create Database' button.

图 5.4 Databases 页面

单击 k8s 数据库右侧的“Explore Data”，在 Query 输入框中输入“list series”，即可查看所有的 series（序列表）。如图 5.5 所示是 Heapster 创建的全部 series。

The screenshot shows the InfluxDB Data Interface under the 'Data Interface' tab. At the top, there are tabs for 'Databases', 'Cluster Admins', and 'Cluster'. The user is connected as 'root' on IP '192.168.1.130'. The main area is titled 'Data Interface' and has a 'Read Points' section. In the 'Query' input field, the text 'list series' is entered. Below the input field is a 'Execute Query' button. The results are shown in a table titled 'list_series_result' with two columns: 'time' and 'name'. The table lists several system metrics, such as 'cpu/limit_gauge', 'cpu/usage_ns_cumulative', 'log/events', 'memory/limit_bytes_gauge', 'memory/major_page_faults_cumulative', 'memory/page_faults_cumulative', 'memory/usage_bytes_gauge', 'memory/working_set_bytes_gauge', and 'uptime_ms_cumulative'. A note at the bottom right of the results area states: 'InfluxDB features a SQL-like query language'.

图 5.5 list series 结果页面

Heapster 中的 metric (性能指标) 数据模型包括:

- ① cpu/limit
- ② cpu/usage_ns
- ③ log/events
- ④ memory/limit_bytes
- ⑤ memory/major_page_faults
- ⑥ memory/page_faults
- ⑦ memory/usage_bytes
- ⑧ memory/working_set_bytes

在保存数据到 InfluxDB 中后, InfluxDB 生成了以下 series:

- ① cpu/limit_gauge
- ② cpu/usage_ns_cumulative
- ③ log/events
- ④ memory/limit_bytes_gauge
- ⑤ memory/major_page_faults_cumulative
- ⑥ memory/page_faults_cumulative
- ⑦ memory/usage_bytes_gauge
- ⑧ memory/working_set_bytes_gauge
- ⑨ uptime_ms_cumulative

对于 series 的命名,如果聚合的是累计值(如cpu 使用时间 ns),则在 serie 名称中用 cumulative 表示; 如果聚合的是瞬时值 (如内存使用字节数), 则在 serie 名称中用 gauge 表示。

我们可以对每个 series 进行 SELECT 操作, 例如查询累计 CPU 的使用时间:

```
select * from "cpu/usage_ns_cumulative" limit 10
```

如图 5.6 所示是 cpu/usage_ns_cumulative 的部分结果及一个图片。



图 5.6 查询 cpu/usage_ns_cumulative 结果页面

5.2.4 Grafana 页面查看和操作

访问任意一台 Node 上 Grafana 的端口 30080，即可查看监控数据的图表展示画面。如图 5.7 所示是 Grafana 的主页，以折线图的形式展示了所有 Node 和全部容器的 CPU 使用率、内存使用情况等信息。



图 5.7 Grafana 主页

Grafana 还提供了基于 Label 的 Pod 的查询，我们可以对某个图表的查询条件进行编辑，如图 5.8 所示。

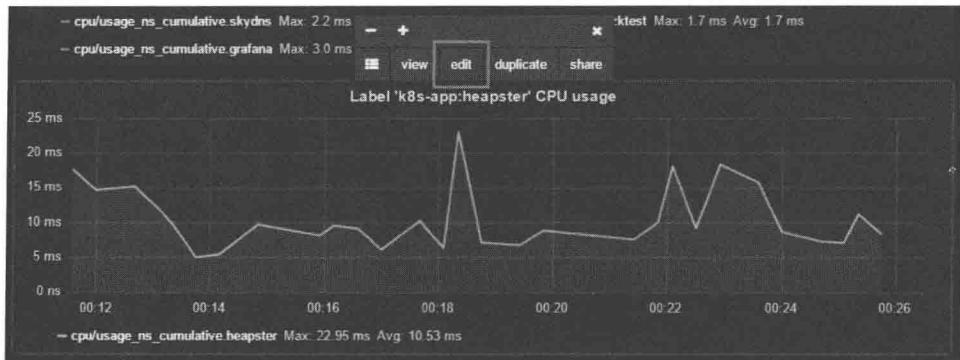


图 5.8 编辑折线图

在编辑页面的 where 条件中可以输入 Label 条件查询所需的 Pod 信息，如图 5.9 所示。

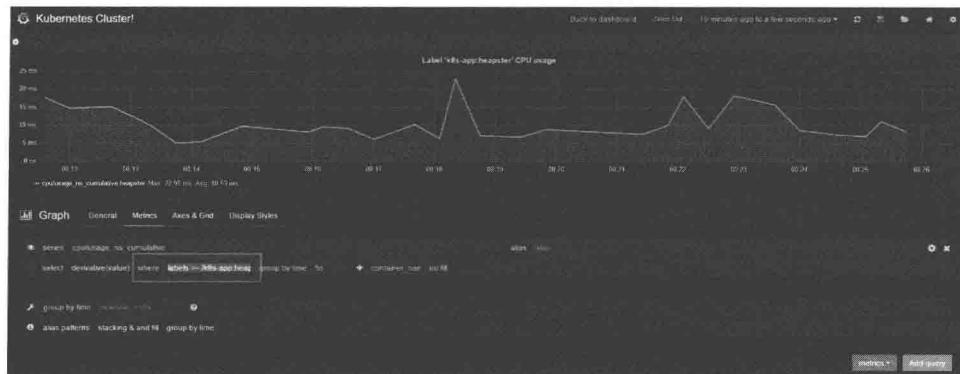


图 5.9 修改 Label 查询条件页面

至此，对 Kubernetes 集群的监控系统就搭建完成了。

在本例中使用了 ServiceAccount、Secret 来保证 Pod 与 master 之间的安全通信；另外，在 Pod 之间使用 DNS 来进行服务的查找——这些都是 Kubernetes 系统中通用和推荐的使用方式。

5.3 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。

5.3.1 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以下网址中获取：

<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 `SeedProvider` 类——`KubernetesSeedProvider` 打包到镜像中了。

定制的 `KubernetesSeedProvider` 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。

5.3.2 通过 Service 动态查找 Pod

在 `KubernetesSeedProvider` 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。如图 5.10 所示描述了

Cassandra 新节点加入集群的过程。

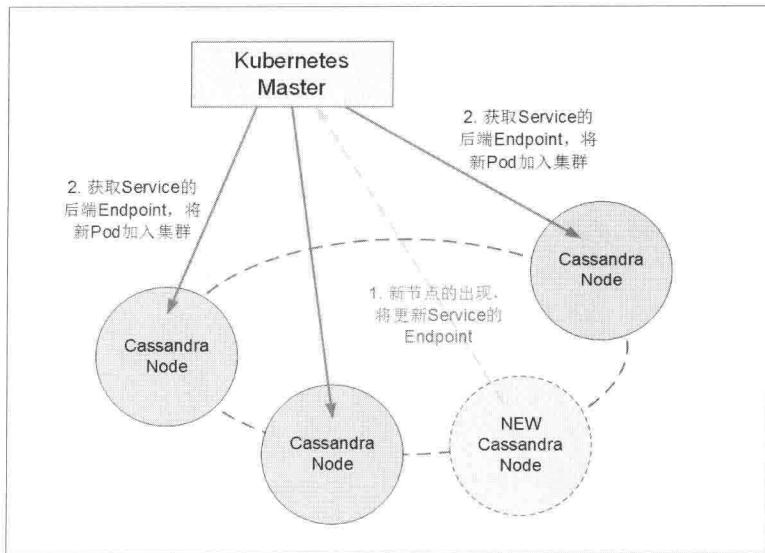


图 5.10 Cassandra 新节点加入集群的过程

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。

5.3.3 Cassandra 集群新节点的自动添加

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 Pods 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 地址即为两个 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，也可以通过查看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。

5.4 集群安全配置案例

Kubernetes系统提供了三种认证方式：CA认证、Token认证和Base认证。安全功能是一把双刃剑，它保护系统不被攻击，但是也带来额外的性能损耗。集群内的各组件访问API Server时，由于它们与API Server同时处于同一局域网内，所以建议用非安全的方式访问API Server，效率更高。

本节将对集群的双向认证配置和简单认证配置过程进行详细说明。

5.4.1 双向认证配置

双向认证方式是最为严格和安全的集群安全配置方式，主要配置流程如下。

(1) 生成根证书、API Server 服务端证书、服务端私钥、各个组件所用的客户端证书和客户端私钥。

(2) 修改 Kubernetes 各个服务进程的启动参数，启用双向认证模式。

详细的配置操作流程如下。

(1) 生成根证书。

用 openssl 工具生成 CA 根证书，请注意将其中 subject 等参数改为用户所需的数据，CN 的值通常是域名、主机名或者 IP 地址。

```
$ cd /var/run/kubernetes  
$ openssl genrsa -out dd_ca.key 2048  
$ openssl req -x509 -new -nodes -key dd_ca.key -subj "/CN=yourdomain.com" -days 5000 -out dd_ca.crt
```

(2) 生成 API Server 服务端证书和私钥。

```
$ openssl genrsa -out dd_server.key 2048  
$ HN=`hostname`  
$ openssl req -new -key dd_server.key -subj "/CN=$HN" -out dd_server.csr  
$ openssl x509 -req -in dd_server.csr -CA dd_ca.crt -CAkey dd_ca.key -CAcreateserial -out dd_server.crt -days 5000
```

(3) 生成 Controller Manager 与 Scheduler 进程共用的证书和私钥。

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

(4) 生成 Kubelet 所用的客户端证书和私钥。

注意，这里假设 Kubelet 所在机器的 IP 地址为 192.168.1.129。

```
$ openssl genrsa -out dd_kubelet_client.key 2048  
$ openssl req -new -key dd_kubelet_client.key -subj "/CN=192.168.1.129" -out dd_kubelet_client.csr  
$ openssl x509 -req -in dd_kubelet_client.csr -CA dd_ca.crt -CAkey dd_ca.key  
-CAcreateserial -out dd_kubelet_client.crt -days 5000
```

(5) 修改 API Server 的启动参数。

增加 CA 根证书、Server 自身证书等参数并设置安全端口为 443。

修改/etc/kubernetes/apiserver 配置文件的 KUBE_API_ARGS 参数：

```
KUBE_API_ARGS=" --log-dir=/var/log/kubernetes --secure-port=443 --client_ca_file=/var/run/kubernetes/dd_ca.crt --tls-private-key-file=/var/run/kubernetes/dd_server.key --tls-cert-file=/var/run/kubernetes/dd_server.crt"
```

重启 kube-apiserver 服务：

```
#> systemctl restart kube-apiserver
```

(6) 验证 API Server 的 HTTPS 服务。

```
$ curl https://kubernetes-master:443/api/v1/nodes --cert /var/run/kubernetes/dd_cs_client.crt --key /var/run/kubernetes/dd_cs_client.key --cacert /var/run/kubernetes/dd_ca.crt
```

注意，API Server 所在主机名为 kubernetes-master。

(7) 修改 Controller Manager 的启动参数。

修改/etc/kubernetes/controller-manager 配置文件：

```
KUBE_CONTROLLER_MANAGER_ARGS="--log-dir=/var/log/kubernetes --service_account_private_key_file=/var/run/kubernetes/server.key --root-ca-file=/var/run/kubernetes/ca.crt --master=https://kubernetes-master:443 --kubeconfig=/etc/kubernetes/cmkubeconfig"
```

创建/etc/kubernetes/cmkubeconfig 文件，配置证书等相关参数，具体内容如下：

```
apiVersion: v1
kind: Config
users:
- name: controllermanager
  user:
    client-certificate: /var/run/kubernetes/dd_cs_client.crt
    client-key: /var/run/kubernetes/dd_cs_client.key
clusters:
- name: local
  cluster:
    certificate-authority: /var/run/kubernetes/dd_ca.crt
contexts:
- context:
    cluster: local
    user: controllermanager
    name: my-context
current-context: my-context
```

重启 kube-controller-manager 服务：

```
#> systemctl restart kube-controller-manager
```

(8) 配置各个节点上的 Kubelet 进程。

复制 Kubelet 的证书、私钥与 CA 根证书到所有 Node 上。

```
$ scp /var/run/kubernetes/dd_kubelet* root@kubernetes-minion1:/home
$ scp /var/run/kubernetes/dd_ca.* root@kubernetes-minion1:/home
```

在每个 Node 上创建/var/lib/kubelet/kubeconfig 文件，文件内容如下：

```
apiVersion: v1
kind: Config
users:
- name: kubelet
  user:
    client-certificate: /home/dd_kubelet_client.crt
    client-key: /home/dd_kubelet_client.key
clusters:
- name: local
  cluster:
    certificate-authority: /home/dd_ca.crt
contexts:
- context:
  cluster: local
  user: kubelet
  name: my-context
current-context: my-context
```

修改 Kubelet 的启动参数，以修改/etc/kubernetes/kubelet 配置文件为例：

```
KUBELET_API_SERVER= "--api_servers=https://kubernetes-master:443"
KUBELET_ARGS= "--pod_infra_container_image=192.168.1.128:1180/google_containers/pause:latest --cluster_dns=10.2.0.100 --cluster_domain=cluster.local --kubeconfig=/var/lib/kubelet/kubeconfig"
```

重启 kubelet 服务：

```
#> systemctl restart kubelet
```

(9) 配置 kube-proxy。

首先，创建/var/lib/kubeproxy/proxykubeconfig 文件，具体内容如下：

```
apiVersion: v1
kind: Config
users:
- name: kubeproxy
  user:
    client-certificate: /home/dd_kubelet_client.crt
    client-key: /home/dd_kubelet_client.key
clusters:
- name: local
  cluster:
    certificate-authority: /home/dd_ca.crt
contexts:
- context:
  cluster: local
  user: kubeproxy
  name: my-context
current-context: my-context
```

然后，修改 kube-proxy 的启动参数，引用上述文件并指明 API Server 在安全模式下的访问地址，以修改配置文件/etc/kubernetes/proxy 为例：

```
KUBE_PROXY_ARGS= "--kubeconfig=/var/lib/kubeproxy/proxykubeconfig --master=https://kubernetes-master:443"
```

重启 kube-proxy 服务：

```
#> systemctl restart kube-proxy
```

至此，一个双向认证的 Kubernetes 集群环境就搭建完成了。

5.4.2 简单认证配置

除了双向认证方式，Kubernetes 也提供了基于 Token 和 HTTP Base 的简单认证方式。通信方式仍然采用 HTTPS，但不使用数字证书。

采用基于 Token 和 HTTP Base 的简单认证方式时，API Server 对外暴露 HTTPS 端口，客户端提供 Token 或用户名、密码来完成认证过程。这里需要说明的一点是 Kubectl 比较特殊，它同时支持双向认证与简单认证两种模式，其他组件只能配置为双向认证或非安全模式。

API Server 基于 Token 认证的配置过程如下。

(1) 建立包括用户名、密码和 UID 的文件 token_auth_file：

```
$ cat /root/token_auth_file
thomas,thomas,1
admin,admin,2
system,system,3
```

(2) 修改 API Server 的配置，采用上述文件进行安全认证：

```
$ vi /etc/kubernetes/apiserver
KUBE_API_ARGS= "--secure-port=443 --token_auth_file=/root/token_auth_file"
```

(3) 重启 API Server 服务：

```
#> systemctl restart kube-apiserver
```

(4) 用 curl 验证连接 API Server：

```
$ curl https://kubernetes-master:443/version --header "Authorization: Bearer
thomas" -k
{
  "major" : "1",
  "minor" : "0",
  "gitVersion" : "v1.0.0",
  "gitCommit" : "843c2c5d65278d57d138ff689b7de5151f058570",
  "gitTreeState" : "clean"
}
```

API Server 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:

```
cat /root/basic_auth_file
thomas,thomas,1
admin,admin,2
system,system,3
```

(2) 修改 API Server 的配置，采用上述文件进行安全认证:

```
vi /etc/kubernetes/apiserver
KUBE_API_ARGS= "--secure-port=443 --basic-auth-file=/root/basic_auth_file"
```

(3) 重启 API Server 服务:

```
#> systemctl restart kube-apiserver
```

(4) 用 curl 验证连接 API Server:

```
$ curl https://kubernetes-master:443/version --basic -u thomas:thomas -k
{
    "major" : "1",
    "minor" : "0",
    "gitVersion" : "v1.0.0",
    "gitCommit" : "843c2c5d65278d57d138ff689b7de5151f058570",
    "gitTreeState" : "clean"
}
```

(5) 使用 Kubectl 时则需要指定用户名和密码来访问 API Server:

```
$ kubectl get nodes --server="https://kubernetes-master:443" --api-version="v1" --username="thomas" --password="thomas" --insecure-skip-tls-verify=true
```

5.5 不同工作组共享 Kubernetes 集群的案例

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 的设置来实现对不同工作组进行区分，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。

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。

5.5.1 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
```

5.5.2 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 `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。

5.5.3 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 2 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-0fq9     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-xyrkk     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，并且它们都能够在同一个 Kubernetes 集群中同时工作。

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, 读者有兴趣可以查看 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 架构的包含 Server 端二进制可执行文件(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, 我们接下来介绍下如何在 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 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，所有的源码已经依赖的源码都在 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。

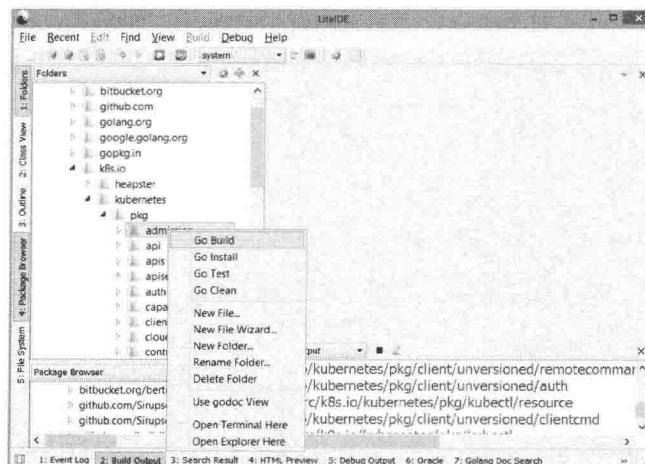


图 6.1 LiteIDE 编译 Kubernetes 的 package

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/GoogleCloudPlatform/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，如接口数据结构相关的 <code>MetaData</code> 结构、 <code>Volume</code> 结构、 <code>Pod</code> 结构、 <code>Service</code> 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，如 <code>v1beta</code> 、 <code>v1</code> 及 <code>latest</code>	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 <code>apiserver</code> 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

续表

package	模块用途	类数量
auth	3A 认证模块，包括用户认证、鉴权的相关组件	少
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，如对 Pod、Service 等的增删改查，这个模块也定义了 KubeletClient，同时为了高效的对象查询，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 TCPLoadBalancer 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 EBS 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，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，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，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、关键代码分析以及设计总结等方面进行深入讲解。

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 (s *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，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 的数据以及上述过程中创建的一些对象 (Kubelet Client、etcd Client、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 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.IPNet
    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外，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，如`/people/{person_id}/groups/{group_id}`；
- ◎ 大大简化了 Rest API 的开发工作；

- ⑤ 底层实现采用 Golang 的 HTTP 协议栈，几乎没有限制；
- ⑥ 拥有完整的单元包代码，很容易开发一个可测试的 Rest API；
- ⑦ Google AppEngine ready。

go-restful 框架中的核心对象如下。

- ⑧ 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；
- ⑨ 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；
- ⑩ 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；
- ⑪ 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）。

```
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
converting 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)
//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.Schema`，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：

- ① `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；
- ② 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；
- ③ `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/etcdb/etcdb.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/etcdb/etcdb.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()`。

- ② 将键值对写入 etcd: e.Helper.CreateObj(key, obj, out, ttl)，在这个方法中通过调用 runtime.Codec 完成从对象到字符串的转换，最终保存到 etcd 中。
- ③ 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.Connecter	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，那么我相信你对于 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；

- ① 不实现 Proxy 代理；
- ② 不做可拔插的权限控制设计（或者根本没有）；
- ③ 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，而有些资源则允许“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。

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/etc/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，你再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
}
```

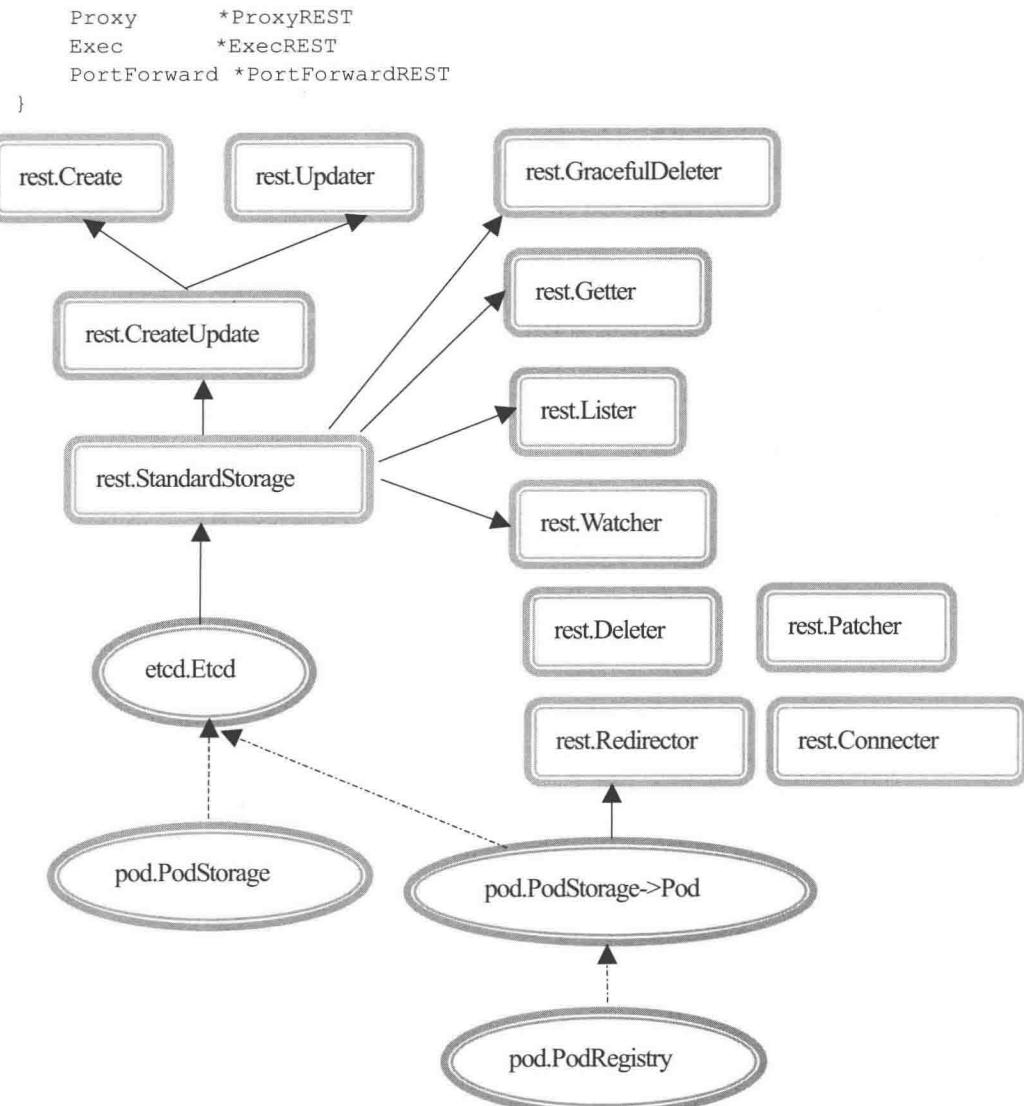


图 6.2 API Server 的 Storage 设计全景图

所以，这里的 PodStorage 应该重命名为 AllPodResStorage，而真正的 PodStorage 其上就是里面的那个 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 Kubelete 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。

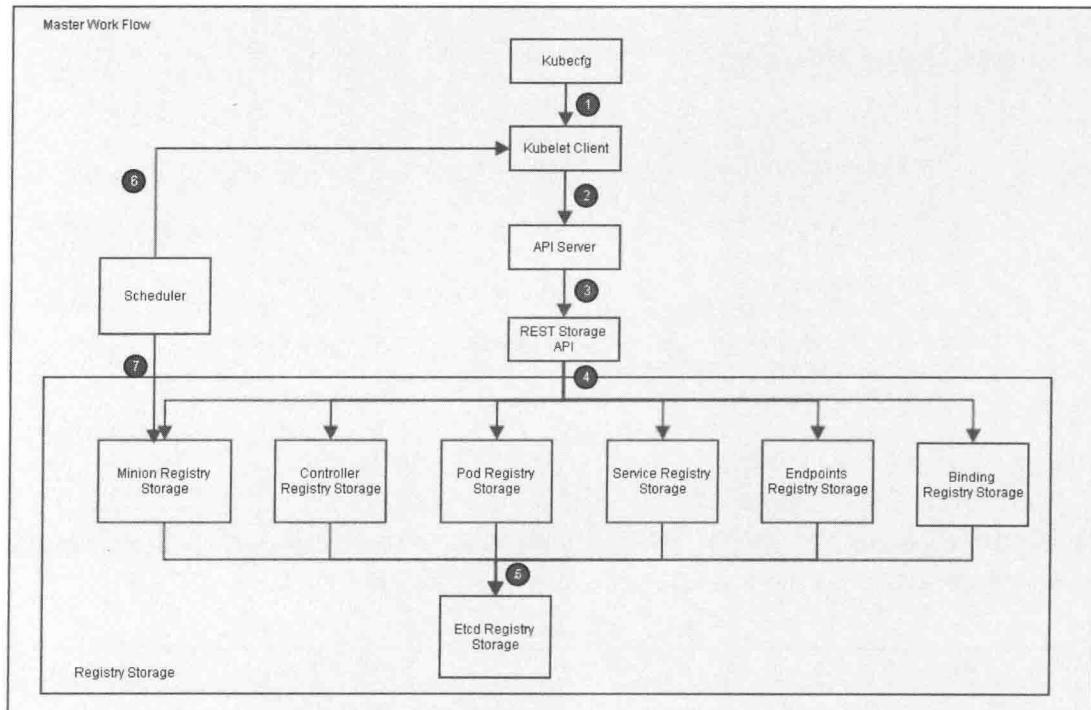


图 6.3 Master 节点数据流图

6.3 kube-controller-manager 进程源码分析

运行在 Master 节点上的第二个进程就是 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.NewCMSServer()
    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 的同步任务数量
NodeSyncPeriod	5 秒	从 CloudProvider 处同步 Node 节点的周期

续表

属性名	默认	含义
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 CMServer 实例并把参数从命令行中传递到 CMServer 后，就调用它的 func (s *CMServer) 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 RCExpectationsManager
    // 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。

- ① 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。
- ② 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）。
- ③ 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。
- ④ 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。
- ⑤ 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。
- ⑥ rcController：framework.Controller 的一个实例，用来实现 RC 同步的任务调度逻辑。
- ⑦ 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。
- ⑧ podStore：类似于 controllerStore 的作用，用来存取和获取 Pod 资源对象。
- ⑨ 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, requeuing 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 同步过程涉及的 Pod 副本数量超过 `burstReplicas` 这个阈值，就会采用 `RCEExpectations` 机制进行限流。`RCEExpectations` 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?以及它与 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。

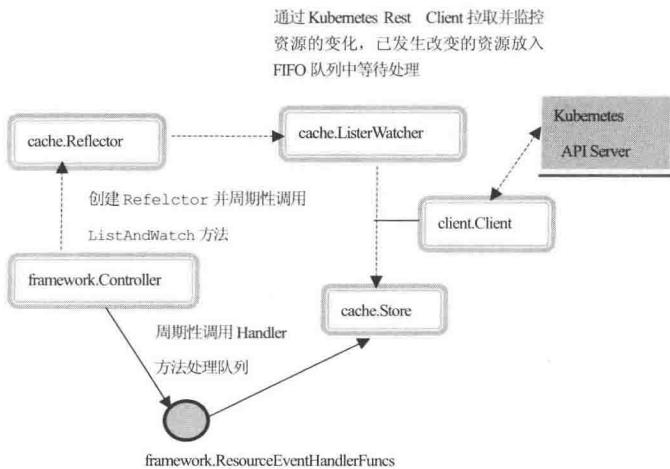


图 6.4 Controller Framework 整体架构设计图

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。

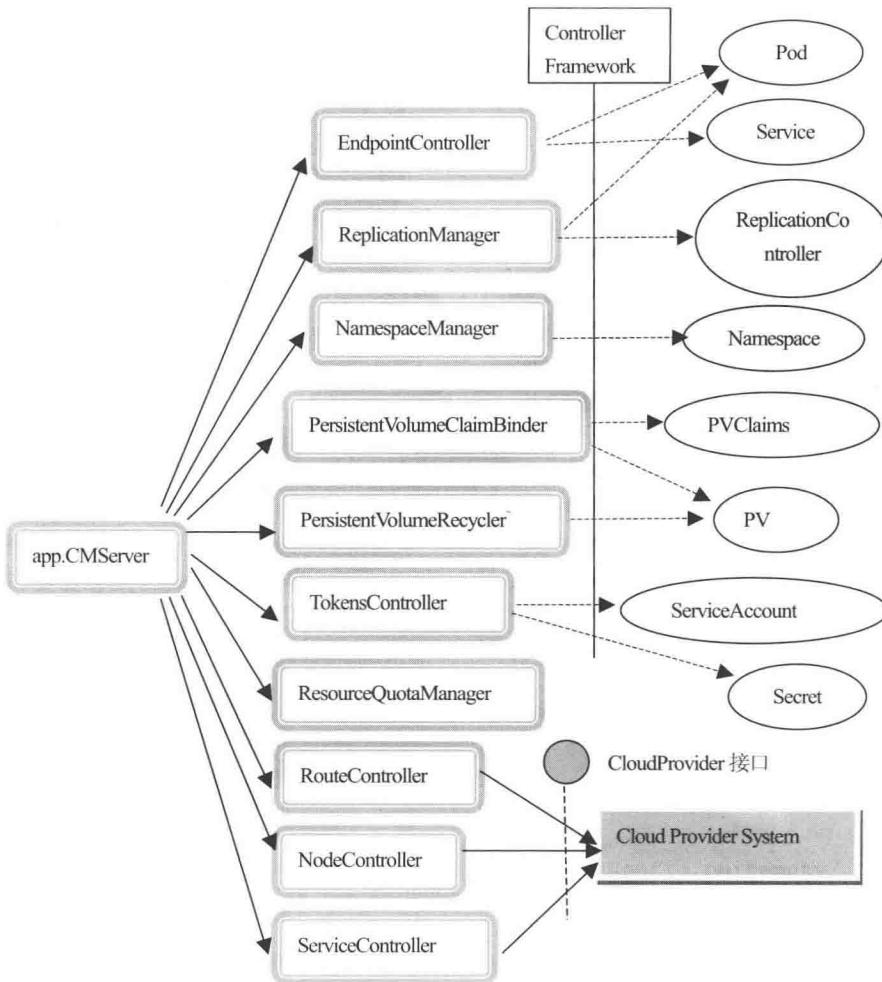


图 6.5 Kubernetes Controller Server 整体设计示意图

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，不太好应用 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 中最优美的代码吧：

```
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！

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。

- ➊ PodQueue：需要调度的 Pod 队列。
- ➋ BindPodsRateLimiter：调度过程中限制 Pod 绑定速度的限速器。
- ➌ 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 SchedulerServer 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就简单明了：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。图 6.6 所示是经典令牌桶流控算法的原理示意图。

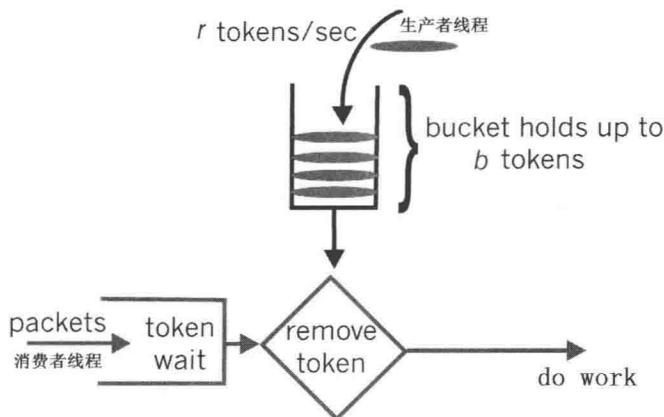


图 6.6 令牌桶流控算法示意图

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外，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，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，图 6.7 所示的 predicates.PodFitsPods 和 priorities.LeastRequestedPriority 就分别使用了上述接口。

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。

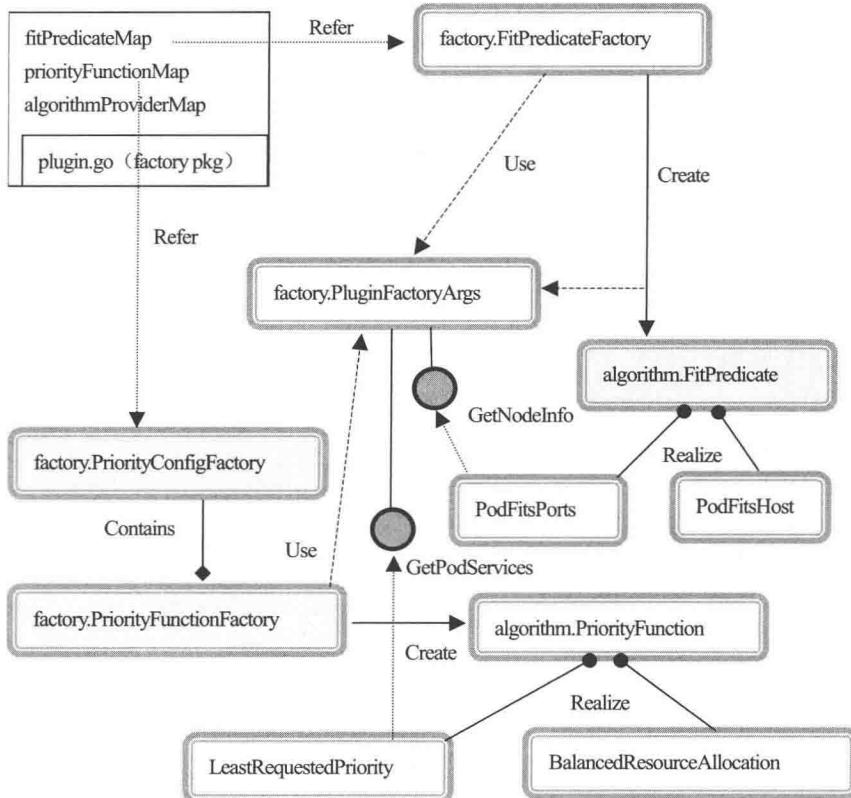


图 6.7 Kubernetes Scheduler Server 调度策略相关设计示意图

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），其实就是一个 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之为“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)
    }
}

```

我们已经是第四次“遇见”这样的代码风格了，代码颜值匹配度高达 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，即每个容器最多保留 2 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、Scheduler 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 Cloud 环境中，则很多环境和资源需要从 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 1.3 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.Recoder = 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 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 的排它锁, 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 创建完成以后被加入 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 中去, 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）及 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())
}

```

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。

- ① 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）。
- ② 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；最后检查磁盘使用情况，如果磁盘可用空间不足，则过滤掉所有 Pod。
- ③ 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.podWorkers）。
- ④ 对于已经删除或不存在的 Pod，通知 podWorkers 删除相关联的 Work Channel（workUpdate）。
- ⑤ 对比 Node 当前运行中的 Pod 以及目标 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。
- ⑥ 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 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，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，则它就会被删除。

◎ 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，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，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 给每个 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 进一步做出审查，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，另外还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 Kuberneate 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"})
    }
}

```

```

        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。

- **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 *TCP SocketAction
}
```

从上面定义来看，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：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)
    return probe.Success, body, nil
}
glog.V(4).Infof("Probe failed for %s, Response: %v", url.String(), *res)
return probe.Failure, body, nil
}
```

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，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.HandleFunc("/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，你经常会有一种感觉——“山穷水尽疑无路，柳暗花明又一村”，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 对象去执行具体的动作，后者通过调用 Dockers Engine 的 API Service 来实现具体功能。

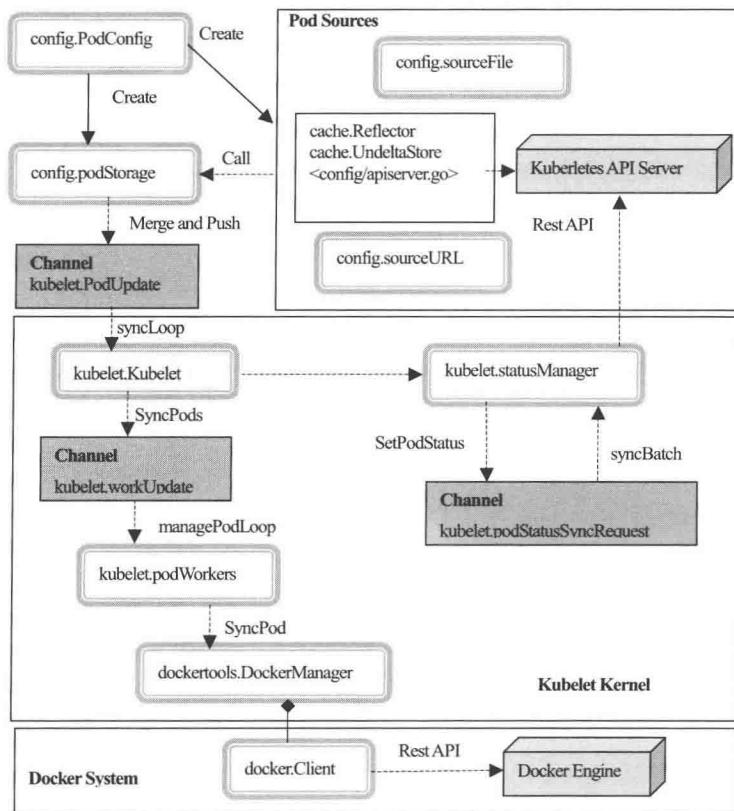


图 6.8 Kubelet 主流程相关的设计示意图

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。

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
```

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 Port 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.Proxy 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 balance` 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.Proxy 是 ServiceConfig 唯一注册的 Listener）的 OnUpdate 接口完成通知。而上述 NewService 就是在 proxy.Proxy 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。

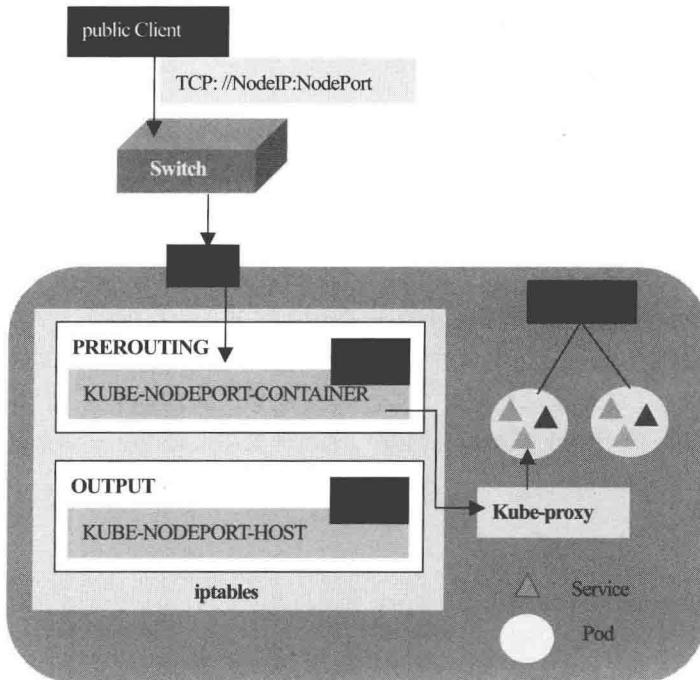


图 6.9 外部应用通过 NodePort 访问 Service 的网络流量示意图

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。

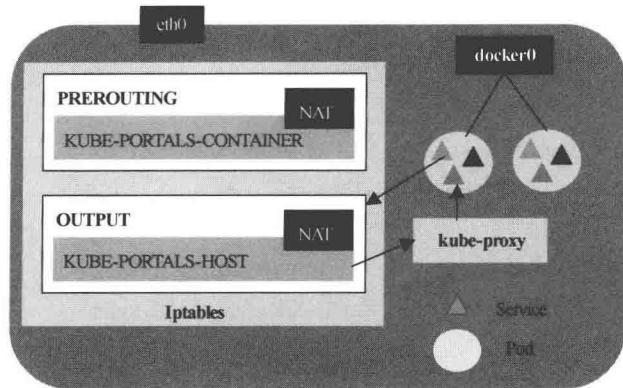


图 6.10 以 Service Portal 方式访问 Service 的流量示意图

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 Endpoints 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。

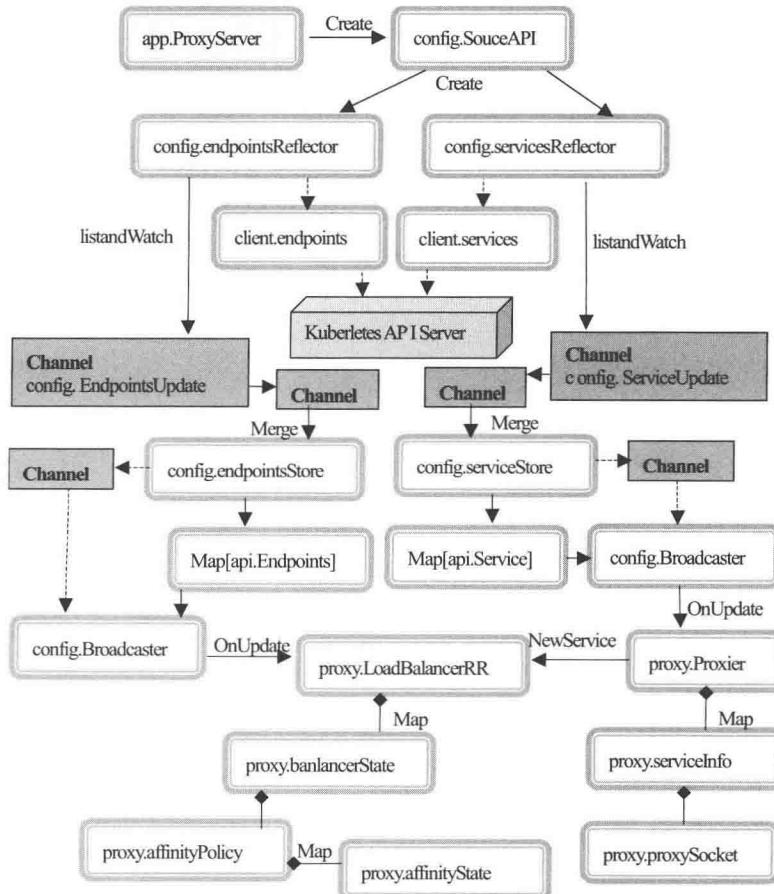


图 6.11 与 Kubelet 总体相关的设计示意图

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](https://github.com/GoogleCloudPlatform/kubernetes/blob/master/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，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处。

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
        }
        fn(ext)
    }
}
```

```
        }
        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)
    }
    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 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-upate 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，我们项目组发起了一个名为 Ku8eye 的开源项目，这是借鉴了 OpenStack Horizon、Clodera Manager 等知名软件的设计思想的一款本土开源软件，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 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。



专家力荐

我相信这是一本到目前为止对从事云计算领域技术实践的人来说非常有价值的书籍。本书作者来自云计算实战一线，敏锐地捕获和探索着各种IT前瞻技术，他们在惠普如日中天的时期加入惠普，是纯粹的技术癖，为世界级的企业构建着相当庞大的信息系统。他们有着全面而扎实的技术架构体系，有着对创新技术天生的热情，有着国际技术领先者的视野，还有着对企业级IT架构的深入把握。

本书囊括了Kubernetes入门、运行机制、原理和高级案例等内容，由浅入深地介绍了当前发展速度极快且被认可度极高的Kubernetes容器云平台，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，有很好的可借鉴性。

不论你是程序员、架构师，还是咨询顾问、IT管理者，你都会通过本书接触到非常热门的Kubernetes和Docker技术的非常清晰、细腻的实践脉络，感受到云计算技术领域的清新气息。

惠普中国区CMS负责人 张红忠

Kubernetes是容器生态圈中的重要一员，发展速度非常快，现在已经拥有500多名代码贡献者。谷歌在容器编排调度方面有着非常丰富的经验，所以Kubernetes的架构设计和理念都很不错。现在，国内已经有很多公司在应用Kubernetes，InfoQ也在这方面发表和策划了很多文章。这是国内专门讲解Kubernetes的重磅开山之作，从架构到源代码、从原理到案例，内容全面而详尽，非常不错。

InfoQ主编 郭蕾

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、缩容等一整套功能。相对于已经很火的Docker，Kubernetes是一款很好的容器管理工具，而本书从Kubernetes的基础、案例到高级话题，都讲得很全面。

随锐科技股份有限公司运维经理 刘成吉

欢迎本书读者和开源爱好者加入Kubernetes专家群：285431657（QQ）



博文视点Broadview



@博文视点Broadview

上架建议：云计算与大数据

ISBN 978-7-121-27639-2



9 787121 276392 >

定价：89.00元



策划编辑：张国霞

责任编辑：徐津平

封面设计：侯士卿

[General Information]

书名=KUBERNETES权威指南 从DOCKET到KURBERNETES实践全接触

作者=龚正，吴治辉，叶伙荣，张龙春，闫健勇等编著

页数=397

SS号=13906035

DX号=

出版日期=2016.01

出版社=电子工业出版社