学号 2017302580098

密级

武汉大学本科毕业论文

云原生容器网络中高可用DNS的设计与实践

院（系）名　称：计算机学院

专　业　名　称：软件工程

学　生　姓　名：姚晓璐

指　导　教　师：贾向阳

二〇二二年八月

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：_____ 日期：_____

摘　　要

云原生是一种构建和运行应用程序的方法，它基于微服务技术、声明式 API 构建，通过容器化封装，部署在容器网络（如 Kubernetes）上，利用容器网络的能力，自动实现服务发现和负载均衡，支持弹性伸缩、动态调度、优化资源利用率。容器网络提供了服务发现能力，可以自动发现上线的服务，并将服务名注册为容器网络中的 IP，以实现不同的部署环境中同样的访问入口。在 Kubernetes 中服务发现是通过 Kubernetes 集群中的域名解析服务（DNS）实现。Kubernetes 提供了 CoreDNS 和 Node Local DNS 两级 DNS 服务，然而这种方案存在单点故障问题，当 Node Local DNS 所在的 Pod 无法工作时，会造成域名解析失败。

因此本文主要工作是研究一种高可用 DNS 方案，解决 Node Local DNS 存在的单点故障问题。该方案在 Node Local DNS 的基础上，采用准入控制器实现，使得在有 Pod 生成时能够自动修改其 DNS 配置，将 CoreDNS 作为备用的 DNS 服务器。在 Node Local DNS 的 Pod 不可用时，DNS 请求能够请求到 CoreDNS，保证 Kubernetes 集群的 DNS 解析能够正常进行。

对比实验证明，在 Node Local DNS 不可用时，Kubernetes 原方案有至少 5 秒的延迟（Pod 重建导致），而本文中的高可用 DNS 只有 2 秒的延迟（查询导致），在高并发环境大大提高了系统的可用性。同时，压力测试也表明，本文的方案在 Node Local DNS 可用情况下，与 Kubernetes 原方案的性能没有差别。

关键词：容器网络；服务发现；DNS 解析；可用性

ABSTRACT

Cloud-native is an approach to build and run applications, which are built on microservices technology and declarative APIs. It is encapsulated by containerization and deployed on a container network (e.g. Kubernetes). Besides, it leverages the capabilities of the container network to automatically implement service discovery and load balancing, support elastic scaling, dynamic scheduling, and optimize resource utilization. The container network provides the capability of service discovery to automatically discover the services that come online and register the service names as IPs in the container network to achieve the same access portal in different deployment environments. Service discovery in Kubernetes is achieved through the Domain Name Resolution Service (DNS) in the Kubernetes cluster. kubernetes provides two levels of DNS services, CoreDNS and Node Local DNS, however, this solution suffers from a single point of failure. When the Pod that the Node Local DNS is located is not working, it will result in domain name resolution failure.

Therefore, the main focus of this paper is to investigate a highly available DNS solution to solve the single point of failure problem of Node Local DNS. The solution is based on Node Local DNS and uses an admission controller to automatically modify its DNS configuration when a Pod is created, using CoreDNS as a backup DNS server. When a Pod of Node Local DNS is not available, DNS requests can be requested to CoreDNS, ensuring that DNS resolution of the Kubernetes cluster can be performed properly.

Comparative experiments demonstrate that the original Kubernetes solution has a latency of at least 5 seconds (due to Pod rebuild) when Node Local DNS is unavailable, while the highly available DNS in this paper has a latency of only 2 seconds (due to queries), which greatly improves system availability in a highly concurrent environment. Also, stress tests show that there is no difference in performance between this paper's solution and the original Kubernetes solution when Node Local DNS is available.

Key words: Container Network; Service Discovery; DNS Resolution; Availability

目　　录

1 绪论

1.1 研究背景

2006 年，云计算概念首次被提出[1]。2013 年，云原生的概念首次被提出，它可以用来更好地解决传统应用迭代过程繁琐、升级缓慢、错误难以准确定位、故障修复困难等问题[2]。2018 年，容器化技术已在超过三分之一的中国企业中运用；到了 2019 年，我国更是有超过六成的企业在生产环境中使用容器部署环境，增长十分迅速[3]。更详细地讲，它是将应用打包成容器镜像，并将镜像部署到 Kubernetes 容器云上运行[4, 5]。Kubernetes 是容器编排调度引擎，其诸多设计法则都与云原生相契合[6]。大多数传统的应用，不需要做任何改动，就可以在云平台上运行[7]。

Kubernetes 的网络模型较为复杂[8, 9]，在服务发现上，Kubernetes 提供了两种技术方式，一种是基于环境变量的方式，另一种是基于域名解析的方式。其中主要使用的方式是域名解析。CoreDNS 是目前 Kubernetes 集群中默认的 DNS 解析服务器，它承担了整个集群的 DNS 解析流量。作为 Kubernetes 集群服务发现体系的核心组件，CoreDNS 负责服务的域名解析，解决了 Kubernetes 集群间服务互相通信的问题。由此可见，域名解析服务器的稳定性和延迟关系到集群内服务间请求的质量，也关系到请求第三方接口的请求质量。所以，在大规模实践过程中需要着重关注其性能和高可用。

1.2 研究动机

在以往，传统应用部署在云服务器上。云服务器是一种基础设施即服务级别的云计算服务[10]，它包含实例、镜像、快存储、网络等功能组件，可以实现应用的快速部署。近两年，出于帮助开发者降低成本、提高服务器资源利用率、提升发布效率等原因，一些 PaaS（Platform-as-a-Service，平台即服务）平台开始鼓励开发者由云服务器向云原生迁移，实际上，也就是计算资源的迁移[11]。迁移过程中，有一种典型的应用类型，它们具有非常高频的云关系型数据库、Redis、以及 QPS 可能达 5 ~ 8 万的开放 API 访问。云关系型数据库，简单理解，它就是一个在线数据库，相应地有容灾、备份、迁移等本地数据库没有的功能，相比起在服务器上自建数据库，它的成本更低。Redis 是一个高性能的键值存储系统，由于是内存高速

缓存，它对于热点数据的读取非常快，适用于缓存、高并发等场景。而这些无一例外都对 DNS 域名解析服务非常依赖。客户在迁移的过程中，出现了一些问题，其中以 PHP 编程语言作为主要语言的应用为主，因为这类应用的框架并没有做很好的缓存，导致 DNS 解析请求几乎都需要 CoreDNS 来完成。初期负载升高，响应时间超长，超时增多，之后出现了大量的 5XX 错误，部分 Pod 重启。经排查发现，这是由于 DNS 解析出现问题，使得部分请求超时失败，从而导致大量请求堆积。帮助客户更好地从云服务器迁移到 Kubernetes，有助于他们更好地进行开发、部署、运行、升级等工作[12]。

当前的迁移过程中还存在一些问题[13]。Kubernetes 网络模型比较复杂，尽管 Kubernetes 本身在很多层面提供了较好的底层网络组件维护功能，但是对于中心化的 DNS 组件来说，整个集群的服务发现需求完全依赖于 CoreDNS[14]。在现有的电商业务系统中，由突发流量带来的众多挑战中，如何保障 CoreDNS 的高可用是一个非常有挑战性的工作。例如，由于高频的数据库访问等对 DNS 解析服务的严重依赖，Kubernetes 现有的解决方案 CoreDNS 无法处理这些流量，这将会导致请求超时、大量请求堆积等问题，可能造成应用宕机，影响生产等后果。

结合具体的实践场景，围绕这个挑战，需要攻克许多技术难点，最终提供出 Kubernetes 容器网络高可用的解决方案，为平台上的云原生应用提供强有力的支撑保障，最终保证整个生态业务的稳定性。在现有的 DNS 体系结构下，由于本地（Pod 所在的节点）并没有 CoreDNS 实例，会导致 DNS 解析需要跨节点请求。在高并发的场景下，CoreDNS 性能容易达到瓶颈[15]。

NodeLocal DNSCache 是 Kubernetes 1.15 发布的新特性之一，它在集群上设置一个守护进程集来存放 DNS 缓存，减少了延迟，提高了集群 DNS 服务的性能和可靠性[16]。它使得每个节点上都有一个 DNS 实例，各个节点上的 Pod 在发出 DNS 请求时，都会请求到各自节点上的 DNS 缓存。如果该缓存无法解析该 DNS 请求，则会将该请求转发到其他 DNS 解析服务器。现有的基于 NodeLocal DNSCache 的产品 Node Local DNS 存在一个主要问题是，当其不可用时，会极大地影响 DNS 解析能力。因为节点上所有 DNS 解析请求都会首先并且必须经过 Node Local DNS Pod，当其失去缓存以及转发能力时，DNS 解析请求都无法得到响应。因此，本文研究的重点是 DNS 的高可用实践。

1.3 研究内容

针对上述问题，本文提出了一种基于 Kubernetes 准入控制器的解决方案，命名为高可用 DNS。当 Node Local DNS 不可用时，它可以将 DNS 请求发送到 CoreDNS，从而使得在 Node Local DNS 恢复的这段时间，DNS 仍然能够正常解析，从而使得服务正常进行。具体实现上，准入控制器对集群中 Pod 的创建动作进行监测，当有监测到时，修改 Pod 的 DNS 配置，将 CoreDNS 作为备用的 DNS 服务器。也就是说，原先使用 Node Local DNS 的情况下，DNS 服务器只有一个，即用来转发 DNS 请求和缓存解析结果的 Pod。添加了 CoreDNS 后，当 DNS 请求从 Node Local DNS 的 Pod 那得不到响应时，就会从 CoreDNS 进行请求。

在电商场景下，依赖 DNS 解析的高频云数据库访问，以及 QPS 高达 5 ~ 8 万的开放 API 调用比较常见。如果 DNS 解析失败，会导致应用响应超时、服务请求失败，甚至可能造成交易失败等产生资损、舆情的严重后果。因此，保证 DNS 的高可用性，对于应用程序的正常运行，用户体验的提升、可能资损的避免，具有重要意义。

1.4 论文结构

第一章是本文的绪论部分，主要介绍了本文的研究背景、研究动机、研究内容，以及 Kubernetes DNS 域名解析发展现状。

第二章介绍了本文的技术基础，包括本文涉及到的 Kubernetes 的相关概念（Pod、工作负载资源、网络模型、服务等）、CoreDNS 的相关介绍，Node Local DNS 的原理简介，准入控制器的介绍。

第三章描述了本文提出的工具的具体设计方法、实现技术。

第四章对 CoreDNS、Node Local DNS，以及本文提出的高可用方案的 DNS 进行了性能测试。

第五章为本文的结论以及展望部分。

2 技术基础

该章节描述了本文所需要的技术基础，包括 Kubernetes 的相关概念，本文研究的主要内容 Kubernetes 的 DNS 架构，以及本文技术实现涉及的准入控制器。

2.1 Kubernetes 相关概念

2.1.1 Kubernetes 结构

简单来说，Kubernetes 是一个软件系统，可以用它来部署和运行容器化的应用。整体结构上它分为 Master（主节点）和若干个 Node（工作节点）。主节点相当于一个控制面板，它含有包括 API 服务器、etcd、调度器、管理控制器等在内的多个组件，用于控制和管理整个集群系统，以保证集群的高可用性。其中，API 服务器公开了一个 REST 端点，用于客户端应用程序、集群组件、用户等来与集群进行通信。工作节点，顾名思义，是一个运行着它容器化的应用程序的机器。在 Kubernetes 上运行应用程序的结构图如图 2.1 所示。

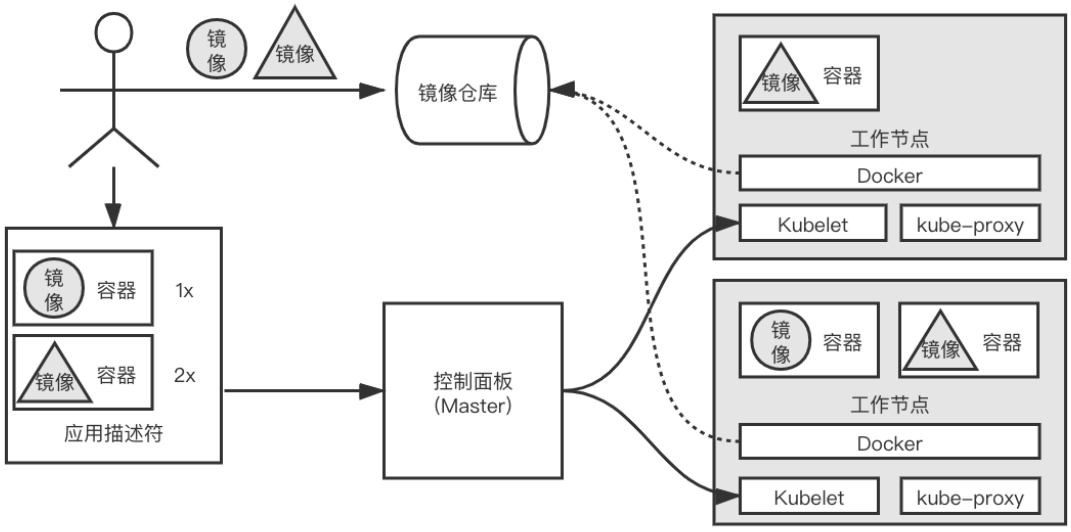

图 2.1 Kubernetes 运行结构图

可以从图 2.1 中看到，用户需要将应用程序打包成镜像，上传至镜像仓库。应用描述符描述了两组不同的容器，也就是把它们分成了两个 Pod（容器组），Pod 是集群中处于运行状态的一组容器。在 Pod 旁可以看到它们并行运行时所需的副本数量。主节点上的 API 服务器会处理这些应用描述，包括对象（如 Pod、服务

等）的状态变更等，它不会去创建 Pod，但只有它才可以直接操作 etcd。接着由调度器根据节点资源、负载均衡等因素分配 Pod 到集群内可用的工作节点上。例如在上图中，圆形镜像所在的 Pod 需要一个副本，三角形镜像所在的 Pod 需要两个副本，API 服务器对其进行处理后，将由调度器对 Pod 进行调度，这里是一个圆形 Pod，两个三角形 Pod，接着，容器将从镜像仓库拉取各自所需的圆形、三角形镜像来运行。

当应用程序成功运行起来时，Kubernetes 会通过探针不断地检测应用程序的部署状态是否和提供的描述相同。比如，需要部署两个三角形实例，但如果其中一个实例出现无法响应或崩溃等停止工作的情况时，那么 Kubernetes 会重启 Pod。如果是工作节点无法正常运行，那么 Kubernetes 会将此工作节点上的所有容器转到可以正常工作的新节点上再运行。

前面已经可以看到，可以通过控制副本数量来修改需要的实例数。当应用程序正常运行时，同样可以对副本数进行水平伸缩，Kubernetes 会完成剩余的调整工作。当然，是否能伸缩成功和集群的资源有关，例如 CPU、内存等。此外，也可以将自动扩缩容的工作交给 Kubernetes 来完成。Kubernetes 会根据当前的指标，诸如 CPU 负载、内存消耗、网络收发速率等指标来对实例数进行动态调整。

分散在不同的工作节点上的应用容器可以对外提供相同或不同的服务，对于提供相同服务的一组容器，Kubernetes 会为其分配一个静态 IP 地址，这样集群中所有的应用程序都能通过这个暴露的地址来访问服务。在外部客户端，也可以通过 DNS 解析找到服务 IP 地址。当然，Kubernetes 组件会将到服务的连接分散在不同的提供此服务的容器上，实现负载均衡。

可以看到，在 Kubernetes 上部署并运行应用程序并不复杂，甚至可以说是非常的简单，此外，它还可以更好地抵御故障风险并进行修复，自动扩容的功能可以使其更好地承担突发的高并发的流量，极大的减少了人工运维成本。

2.1.2 工作负载资源

Kubernetes 内有多种工作负载资源，本文主要介绍 Deployments 和 ReplicaSet、DaemonSet（守护进程集）。由于本文中的测试应用是无状态应用，也就是说它不依赖本地运行环境，实例间互不依赖，可以自由伸缩，因此可以用 Deployments 来管理。一个 Deployment 描述了 Pods 和 ReplicaSets 的期望状态，例如 Pod 需要拉取的镜像、所需要的 CPU、内存等资源，DNS 策略等。ReplicaSet 通过选择算符、

副本个数、Pod 模版来控制 Pod 的副本数量。当有 Pod 被删除时，会自动创建新的 Pod，以保证 Pod 数量达到期望值。

DaemonSet 也叫守护进程集，它会确保每一个节点或某些节点上运行一个 Pod 的副本，本文中提到的 Node Local DNS 就是使用的 DaemonSet。它的拓扑结构如图 2.2 所示，不管目标节点增多或减少，它总是保证节点上都有这个 Pod 的副本，这也是之后测试 Node Local DNS Pod 删除并恢复运行这个过程时间的依据，即删除了 Pod 后它会立即在该节点上再次自动创建，只是这中间会有一定的时间消耗，无法保证时刻可用。

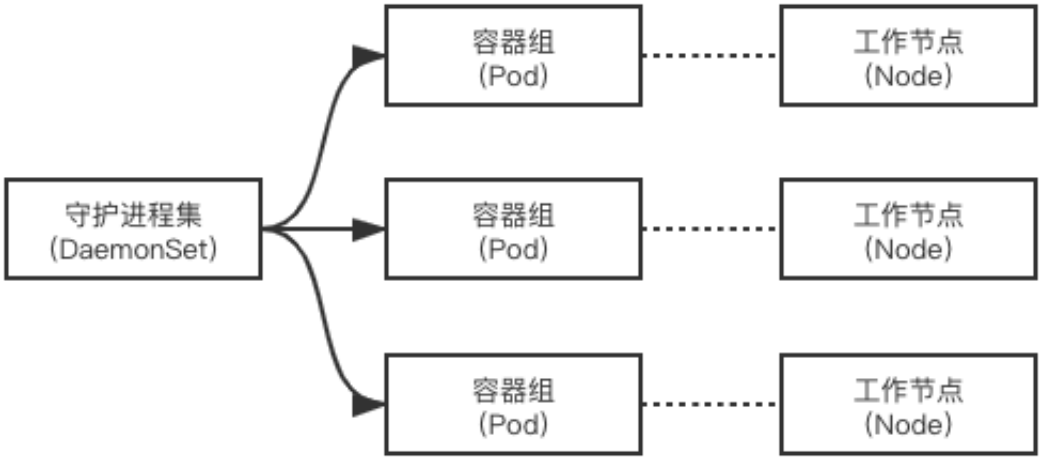

图 2.2　DaemonSet 拓扑结构图

2.1.3　服务发现

Kubernetes 会为每个 Pod 分配单独的 IP，但是，由于 Pod 随时可能启动或关闭，会导致 IP 会不停地变化；此外，当 Pod 进行水平扩缩容时，这一组内的多个 Pod 会对外提供相同的服务。对于客户端来说，它不能提前知道 Pod 的 IP 地址，也并不需要知道 Pod 的数量，它只需要知道这个服务的地址即可。因此，Kubernetes 提供了一种叫做 Service 的资源类型，也就是服务，它使得客户可以通过服务提供的 IP 和端口号来访问具有这个功能的 Pod，而无需顾忌 Pod 的数量或某个 Pod 是否可用，这样就使得开发运维更加便利，并且可以实现负载均衡。

那么问题来了，客户端怎么知道这个服务的 IP 和端口号？这里就涉及到了服务发现，其中一种方式是域名解析。在集群的 kube-system 命名空间下，有一个叫做 kube-dns 的 Pod。这个 Pod 运行着 DNS 解析服务，集群中所有其他的 Pod

的 DNS 都将使用其运行。从原理上来说，Kubernetes 修改了每个容器的/etc/resolv.conf 文件，使得集群内所有 Pod 上的 DNS 查询进程都将使用自身的 DNS 服务器。CoreDNS 即是 Kubernetes 1.12 以来默认的 DNS 服务器。注意，可以通过配置 DNS 策略来决定是否采用 Kubernetes 内部的 DNS 服务器。

2.1.4 网络模型

前文已经提到了 Pod 和服务间的通信以及外部和服务间的通信。Pod 间的通信涉及到多种类型，如图 2.3 所示，可以看到，基本上分为三种，为了更便利地进行通信，现在已经有多种网络接口（CNI）插件可以实现，例如 Flannel、Terway 等[17]。

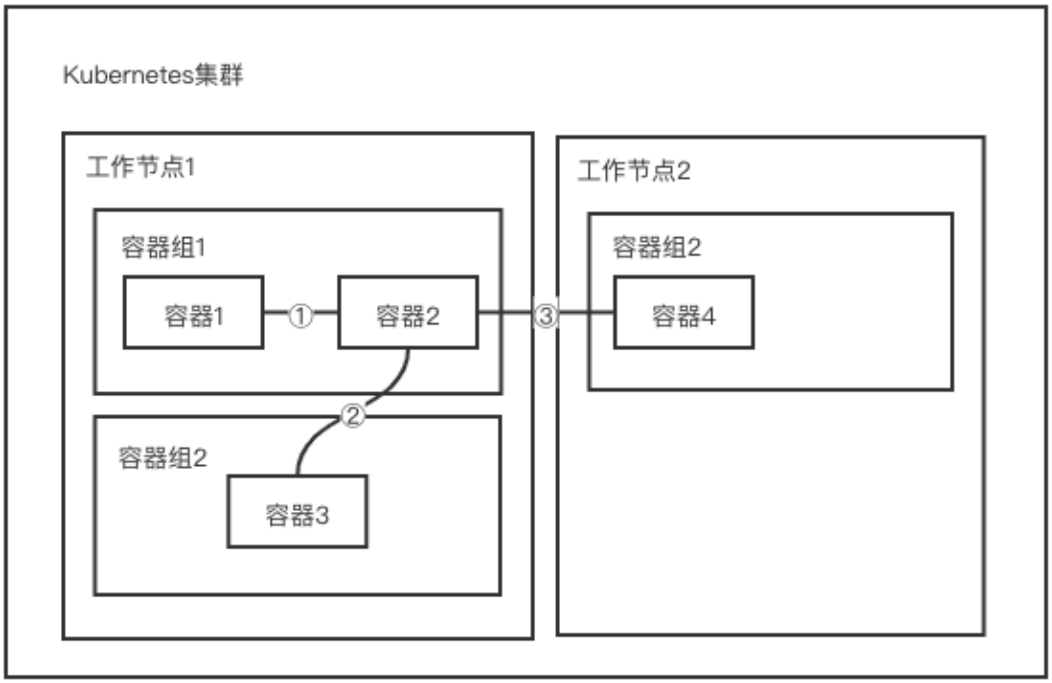

图 2.3 Pod 间通信种类

Flannel 是一种“覆盖网络”，它为集群里不同节点上的 Pod 都分配了一个唯一的、虚拟的 IP 地址，使得不同节点上的容器可以互相通信[18]。其跨节点通信方式如图 2.4 所示，该图示例了 Node1 上 IP 为 10.1.15.2 的 Pod1 和 Node2 上 IP 为 10.1.20.2 的 Pod2 通信的过程。flannel.0 网桥在收到数据包后，它会将其再重新发送出去并发出 ARP 请求，这时，flanneld 程序收到 ARP 请求后，会找到 Node2 上的 flannel.0 的 MAC 地址，并将此地址放入 Node1 host 的 ARP Cache 表中。接着，flannel.0 会将请求二次封装成 UDP 包，并由 VXLAN 隧道发送到 Node2 的 eth0 上。再进行拆包、转发等过程，就可以与 Pod2 内的某个容器进行通信了。

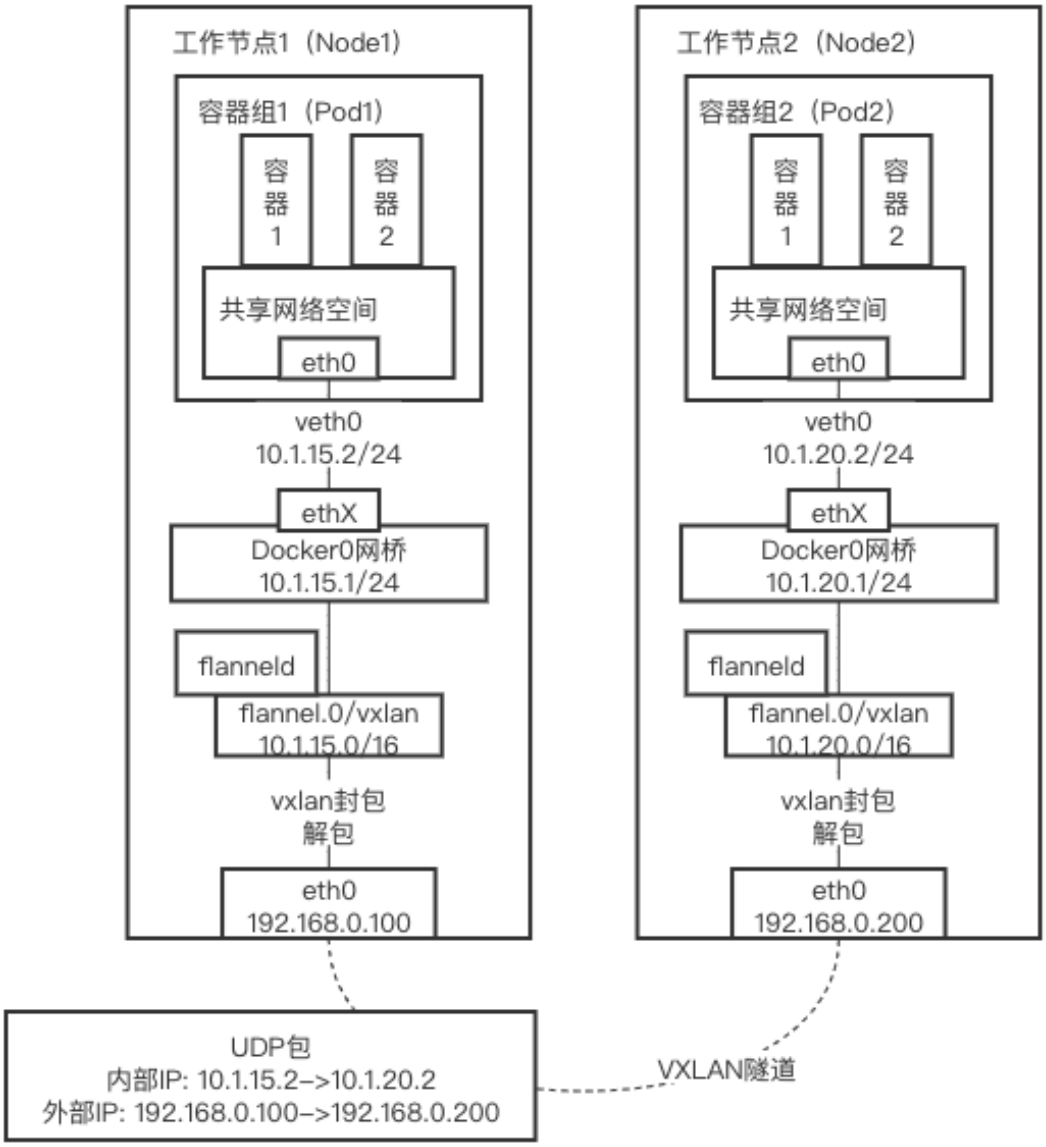

图 2.4　跨节点通信方式

2.2　Kubernetes DNS 架构

2.2.1　CoreDNS

在集群中，CoreDNS 可能有一个或多个副本，它们可能在一个或多个节点上。使用 CoreDNS 时，DNS 请求可能存在跨节点访问，即便发出 DNS 请求的 Pod 所在的节点上有 CoreDNS，也可能会请求到其他节点的 CoreDNS 上，这是因为服务只能绑定相同的 IP，但无法指定到某个特定的 Pod 上（当这个 Pod 存在多个副本时），是由 Kubernetes 决定的。一个简单的例子可以看图 2.5。

可以从图中看到，当 CoreDNS 只有一个副本时，它可能被 Kubernetes 随机调

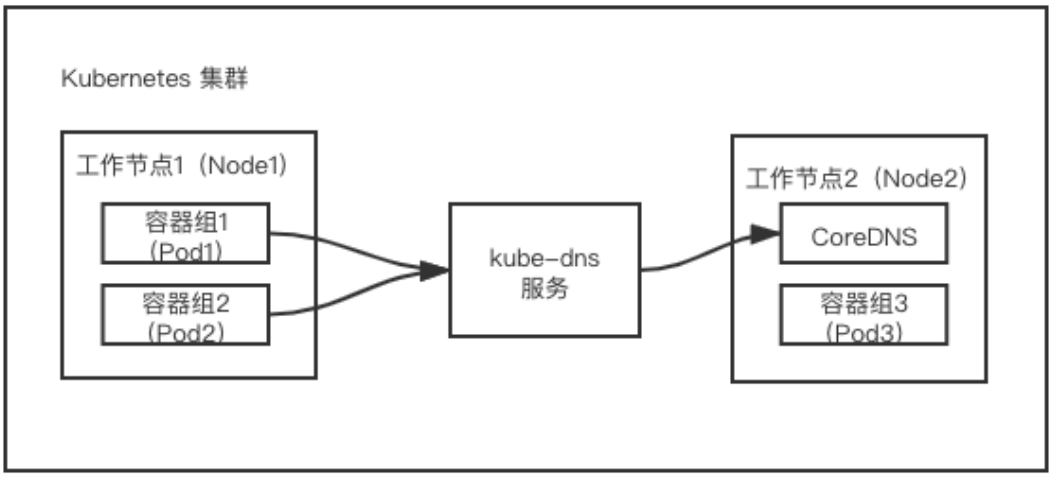

图 2.5　跨节点访问示意图

度到某个节点上,这里 CoreDNS 在 Node2 上。如果此时 Node1 上的应用程序 Pod1 和 Pod2 需要请求 DNS 服务，那么就不得不跨节点请求到 Node2 上，具体的请求过程可以见 2.1.4 小节，存在多次解包、封包过程。

Kubernetes 内域名解析依赖容器里的/etc/resolv.conf 配置文件，此文件的一个例子如图 2.6 所示。

```
nameserver 172.21.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

图 2.6　resolv 文件结构

这里配置的 DNS 服务器是集群中 kube-dns 服务的 ClusterIP，也就是 nameserver 属性的值。集群内所有域名的解析，都要经过此 IP 进行解析，不论是内部域名还是外部域名，并且都需要走图 2.6 中的 search 域。例如，当执行命令“curl a”时，会选择 nameserver 的 IP 进行解析，然后，带入 search 域，进行 DNS 查找，分别是，a.default.svc.cluster.local，a.svc.cluster.local，a.cluster.local，a.local，a。依次执行，直到解析成功为止。这里 options 属性内 ndots 为 5 代表，如果查询的域名包含的点“.”不到 5 个，则会首先使用 search 域的非完全限定名称。例如请求“curl a.b.c.d.e.f”，这里的点有 5 个，那么 DNS 查找顺序将会是，a.b.c.d.e.f，一次查找成功。从中可以发现，假如请求的是外部域名，并且点数小于 5，则将会浪费 4 次查询，降低了 CoreDNS 查询的效率。

2.2.2 Node Local DNS

Node Local DNS 是使用 DaemonSet 实现的，它会在每个节点上都运行一个用于 DNS 缓存的 Pod，示意图见图 2.7。

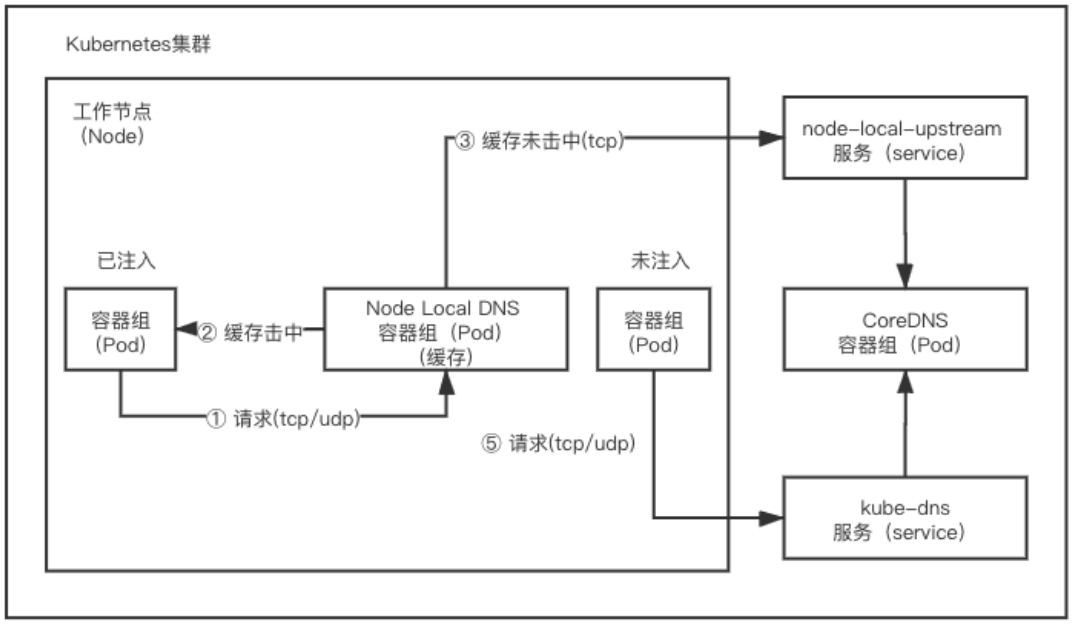

图 2.7　Node Local DNS Pod 示意图

可以看到，在 Kubernetes 集群中，每一个 Node 上都有一个 Node Local DNS Pod。在 2.2 节中提到，即使是 Pod 和 CoreDNS 在同一节点，Pod 也可能会请求到其他节点上的 CoreDNS 进行解析。但是在 Node Local DNS 这里，Pod 发出的请求一定会请求到它所在的节点上的 Node Local DNS，也就是说，在图 2.7 中，Pod1 一定会请求到 Node1 上的 Node Local DNS Pod，Pod2 一定会请求到 Node2 上的 Node Local DNS Pod。这是因为 Node Local DNS 使用了一个叫做 bind 的插件，该插件通过修改 iptables 或 IPVS（IP Virtual Server），使得通过 nameservers 查找时一定会指向该节点上的 DNS 缓存。以 iptables 为例，bind 插件可以修改 iptables 的规则，使得请求到 nameserver 的 IP 的 DNS 请求都转发给该节点上 Node Local DNS Pod 的 IP。因此，各 Pod 即使是在请求相同的 IP，也会请求到特定的 DNS 缓存上，使得 DNS 请求不用再跨节点。

使用 Node Local DNS 请求的流程如图 2.8。

可以在图 2.8 看到，未安装 Node Local DNS 时（第 ⑤ 步），Pod 发出的 DNS 请求会到集群的 kube-dns 服务，由 CoreDNS 处理，在此过程中可能存在跨节点访

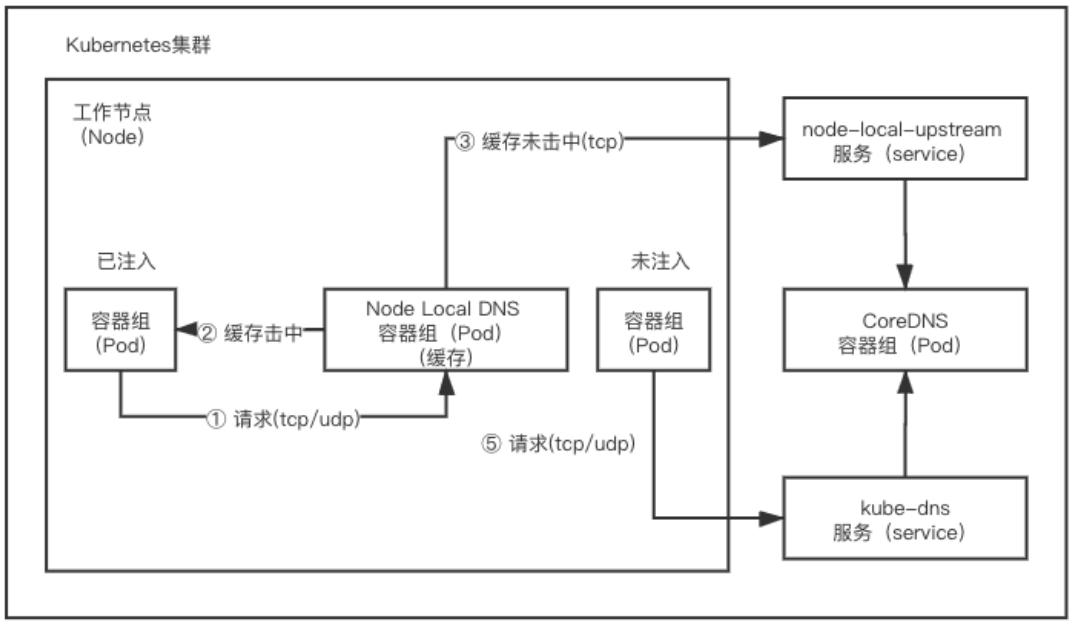

图 2.8　Node Local DNS 架构图

问。安装了 Node Local DNS 后（第①、②、③步），DNS 请求会到同节点的 Node Local DNS Pod 上，该 Pod 缓存了 DNS 解析结果，如果命中，会直接将解析结果返回给请求的 Pod；如果未命中，则会通过 node-local-upstream 服务请求到集群中的 CoreDNS，由 CoreDNS 进行解析后返回。

2.3　准入控制器

前文提到，API 服务器用于集群和客户端应用程序间进行通信，具体的请求过程如图 2.9 所示。

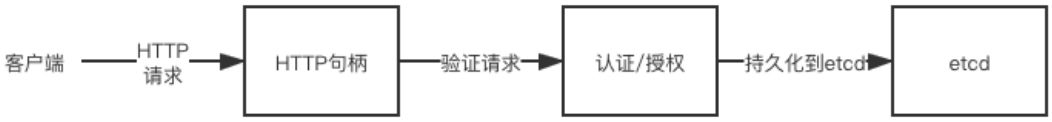

图 2.9　请求流程

从图 2.9 可以看到，客户端应用程序发送 HTTP 请求，经过身份验证并成功授权后，API 服务器能够根据诸如 Pods、Deployments、Namespace 等对象和 Create、Delete 等 HTTP 动作来执行操作，之后会对 etcd 数据存储进行修改并保存，操作完成后，API 服务器会向客户端发送响应。整个请求流程发生在图 2.1 中从控制面板至调度到节点之间。

现在，有这么一种情况，在身份认证和授权成功后、写入 etcd 之前，需要对该请求进行拦截，执行一些操作，例如，拦截该请求，请求得到结果后决定是否要写入 etcd 等。这其实就是 Kubernetes 准入控制器的情况[19]，有了这种插件之后，整个请求流程如图 2.10 所示。

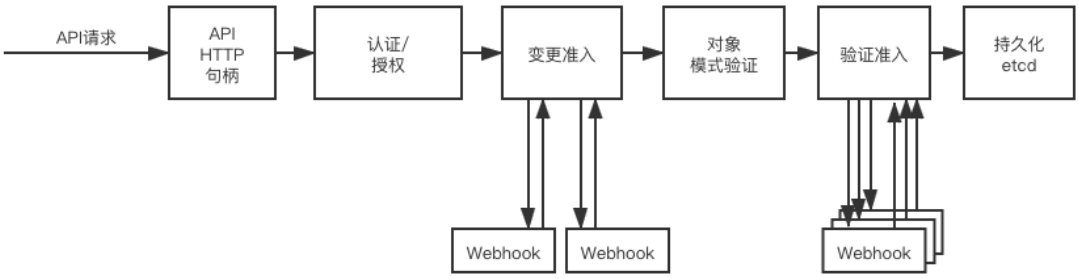

图 2.10　准入控制器请求流程

实际上准入控制器属并不实现任何决策，它将逻辑与 Kubernetes API 服务器进行解耦，使得开发人员可以自定义要执行的逻辑，也就是说，准入控制器实际上是一段代码，里面是开发人员编写的自定义逻辑。由上图中可以看到，准入控制器执行了 Mutating 或 Validation 两种操作类型，也就是变更或验证。按顺序来看，准入控制器在对象持久化之前对到达 API 服务器的请求进行拦截，进行身份验证并授权后依次进行了变更、验证两个阶段。

Kubernetes 内的很多操作都由准入控制器完成，举个例子，NamespaceLifecycle 准入控制器不允许删除 kube-system 等系统层面的命名空间，此外，它还不允许在不存在或正在删除的命名空间内创建新对象，它的存在非常重要，因为要知道如果命名空间被删除，那这个命名空间内的所有对象（包括 Pods、Service、Deployments 等）都会被删除，Kubernetes 的一些系统功能可能就会出现异常，例如删除 kube-system 命名空间，那么 CoreDNS 也会被删除，将导致整个集群的 DNS 解析受影响。其中有两个特殊的控制器是 MutatingAdmissionWebhook 和 ValidatingAdmissionWebhook，它们可以根据 API 中的配置，分别执行变更和验证操作。但值得注意的是，变更 Webhook 可能会对内建资源造成破坏性的修改，也可能会使用户感到疑惑，因为创建的对象和返回的对象不相同。

使用准入控制器可以对对象进行某些修改或验证，例如，可以在 Pod 被 Master 调度前修改 Pod 的某些配置，等等。

3　高可用 DNS 的设计与实现

该章节描述了本文研究内容目前存在的问题，解决问题的实现思路以及实现技术和过程。

3.1　Kubernetes 现有 DNS 的问题

从前文 Node Local DNS 的结构图中不难发现，所有的 DNS 请求都会经过节点中的 DNS 缓存 Pod，只有当缓存未击中时才会请求到服务 node-local-upstream，再从 CoreDNS 获取结果。不难发现，当 Node Local DNS 这个 Pod 不可用时，比如被删除或 Pod 镜像拉取失败，在 Pod 重新创建并能够正常运行之前这段时间内，所有的 DNS 请求都会失败，这就导致了这段时间内应用程序可能会突然响应超时，带来不必要的损失。具体见图 3.1。

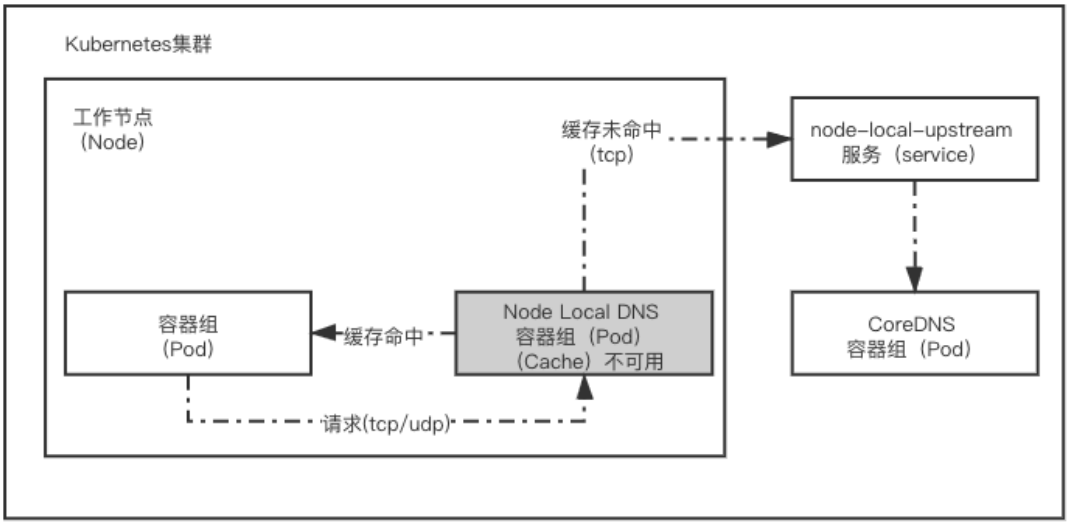

图 3.1　Node Local DNS 不可用示意图

可见，当 Node Local DNS Pod 不可用时，此时 DNS 请求只能面临解析失败的选择。

3.2　高可用 DNS 的设计

针对此问题，本文设计出了一种高可用的 DNS，它可以在 Local DNS 缓存不可用时，将请求直接发送到 CoreDNS，当缓存再次恢复时，又会将请求发送到缓

存的 Pod 上，以此来保证 DNS 的正常解析。同时，针对不同的命名空间，可以自定义采用高可用 DNS 或只使用默认的 CoreDNS。具体的结构如图 3.2 所示。

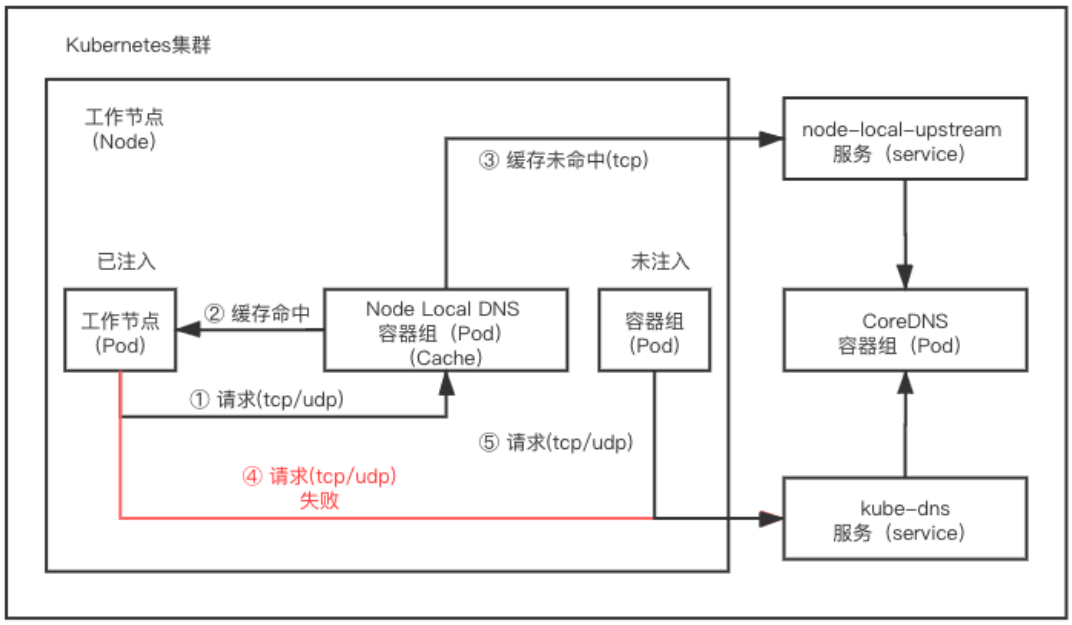

图 3.2　高可用 DNS 架构图

可以从图 3.2 中看到，针对未注入 DNS 本地缓存的 Pod（第 ⑤ 步），其 DNS 请求会直接通过 kube-dns 服务到 CoreDNS；对于注入 DNS 本地缓存的 Pod（第 ①、②、③、④ 步），DNS 请求会首先请求到 Node Local DNS，如果命中则直接返回，未命中则通过 node-local-upstream 服务请求到 CoreDNS。特别的是，当 Node Local DNS 不可用时（第 ④ 步），它将会像未注入 DNS 本地缓存的 Pod 一样，直接请求 CoreDNS。

实质上，对于未注入高可用 DNS 的 Pod，它与单纯使用 CoreDNS 的情况一样，这不属于此次论文工作的内容。论文的关键工作是上图中的第 ④ 步，当缓存不可用时，注入了高可用 DNS 的 Pod 会请求到 CoreDNS 服务器。

3.3　高可用 DNS 的实现技术

3.3.1　技术方案选择

为了实现上图中的设计，首先想到的是修改 DNS 配置。正如在 2.2 节中提到的那样，为 Pod 新增一个备用的 DNS 服务器，也就是说 resolv.conf 配置文件里的 nameserver 需要再增加 CoreDNS 服务器。2.1 节里提到，Pod 会根据应用描述符来创建，而这里并没有办法直接去修改用户的描述文件。但应用描述符会被控制

面板调用，写入 etcd，调度器会将 Pod 分散到节点上。在 Pod 被调度之前，可以对其配置进行修改，也就是使用 MutatingAdmissionWebhook 变更准入控制器，在 yaml 或 JSON 文件生效前进行变更。这样也无需用户进行额外的操作，使得用户体验更好。之后使用 ValidatingAdmissionWebhook 对修改过的内容进行验证。在具体的变更操作上，有两种方式，一种是直接修改 DNS 配置，为其增加备用的 nameserver；另一种是通过文件挂载的方式，将需要的 DNS 配置文件挂载到 Pod 下。这里选择的是第一种，直接修改 DNS 的配置。

3.3.2 整体实现

通过配置将 Webhook 注册到 Kubernetes 中，同时，自定义地监听事件，监听到 Pod 创建事件时，会第一时间调用到 Webhook 内，Webhook 中的方法会对其进行一定的处理，返回结果就是要修改的部分，也就是说，在 yaml 文件生效前进行修改。具体的流程如图 3.3 所示。

在具体实现上，采用 Go 语言编写 MutatingAdmissionWebhook 和 ValidatingAdmissionWebhook。首先 API 请求到达 Kubernetes API 服务器，经过认证并授权后执行 MutatingAdmissionWebhook，具体流程可见 2.3 小节。它针对 dnsconfig-injector=enabled 标签的命名空间生效，针对生效的命名空间里新建的 Pod，如果其注解上带有 dnsconfig-injector-webhook.scom.com/inject: skip，则不会进行修改。接着，通过 API 服务器获取到 kube-system 命名空间下的 kube-dns 服务的 ClusterIP，添加到新建的 Pod 的 DNS Config 的 nameservers 属性中，增加了 CoreDNS 作为备用服务器，以最大程度的保证业务 DNS 请求高可用。最后，将修改了 DNS 配置的 Pod 打上注解 dnsconfig-injector-webhook.scom.com/status: injected。

接着执行 ValidatingAdmissionWebhook。ValidatingAdmissionWebhook 会对前面 MutatingAdmissionWebhook 修改过的 DNS 配置进行验证，除此之外，还会验证修改过的 Pod 是否打上了注解 dnsconfig-injector-webhook.scom.com/status: injected，如果都验证成功，则通过验证，最后可以将数据写入 etcd。

3.3.3 认证/授权

Kubernetes API 服务器本质上是一个 web 服务器，因此也会具有认证/授权机制，不同的是，API 服务器还提供了准入检查。

认证（Authentication）用于识别用户身份，方式有很多，例如 TLS，HTTP token，Service Account 等，这里使用的认证方式是 TLS。授权（Authorization）方式也比

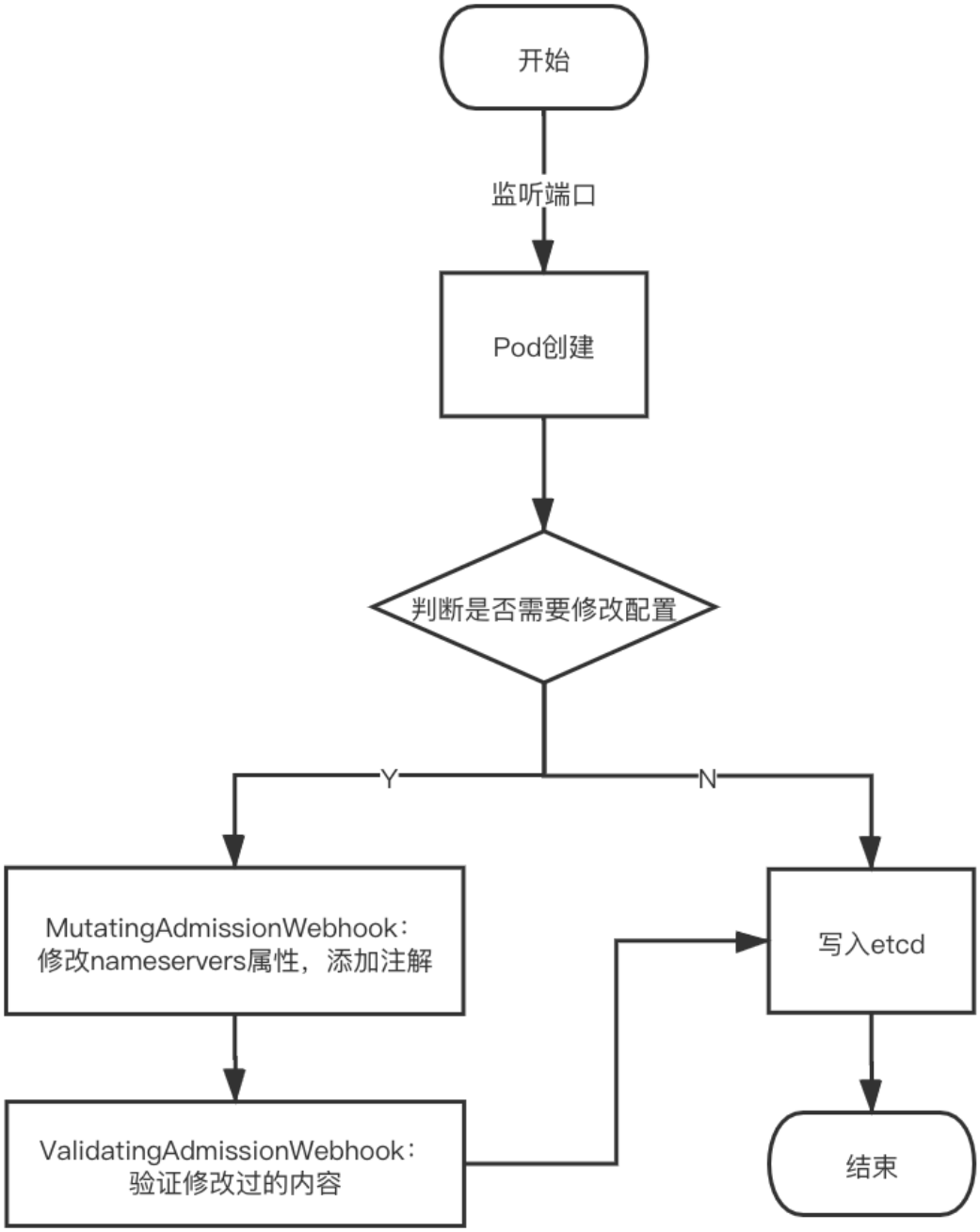

图 3.3　Webhook 流程图

较多，这里使用的是 RBAC，它基于角色授权，使用集群级别的资源需要使用资源对象 ClusterRole。简单理解，它用于描述用户和集群资源之间的连接。论文使用到的集群中具体的角色配置文件如图 3.4。

在证书上，使用脚本完成。脚本使用 Kubernetes 的 CertificateSigningRequest API 生成一个适用于本文中 Webhook 的 Kubernetes CA 签名的证书服务，服务器密钥/证书 Kubernetes CA 证书存储在 Kubernetes 密钥中。首先清空服务器中已经

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: default-role-binding
subjects:
- kind: ServiceAccount
  name: default
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

图 3.4　授权配置文件

创建过的证书请求文件（Certificate Signing Request，CSR），然后创建服务器证书签名请求对象发送到 Kubernetes API，接着，对创建的 CSR 和签名证书进行验证，最后，用 CA 证书和服务器证书/密钥生成 secret，以供后面的 Webhook API 用。接着，同样是用脚本生成 mutatingwebhook-ca-bundle.yaml，其中的 caBundle 来源于前一步生成的值，具体的配置如图 3.5。

可以看到,这里的服务请求路径是“/mutate”,针对资源类型为 Pods 的 CREATE 操作生效，同时，要求其匹配标签 dnsconfig-injector=enabled。caBundle 的值这里并没有写入。

3.3.4　变更

即便是相同的服务名称，不同的集群其 IP 也可能不一样，因此需要动态地获取服务 IP。Kubernetes 提供了 API 供编程方式访问。ctrl.GetConfigOrDie() 方法创建了一个*rest.Config 对象来和 API Server 对话，通过这个对象可以动态地获取到 kube-system 命名空间下 kube-dns 服务的 IP 地址。在 Webhook 启动时，会加载服务端口号、证书和密钥文件、DNS 配置文件。DNS 配置文件前文已对其进行描述，主要由 nameservers、searches、options 三个部分组成。这里初步传入的配置文件如图 3.6 所示。

DNS 配置文件会先被加载在内存中，转换成对应的*v1.PodDNSConfig 结构，具体可见图 3.7，此结构各属性内容刚好对应配置文件各部分，可以理解为将文件内容转换成了内存对象。

此时，就可以对其进行修改，将上述获取到的 kube-dns 服务的 IP 地址追加

```yaml
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: dnsconfig-injector-webhook-cfg
  labels:
    app: dnsconfig-injector
webhooks:
  - name: dnsconfig-injector.scom.com
    clientConfig:
      service:
        name: dnsconfig-injector-webhook-svc
        namespace: kube-system
        path: "/mutate"
      caBundle:
    rules:
      - operations: [ "CREATE" ]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: ["pods"]
    namespaceSelector:
      matchLabels:
        dnsconfig-injector: enabled
```

图 3.5　mutatingwebhook-ca-bundle.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: dnsconfig-injector-webhook-configmap
  namespace: kube-system
data:
  dnsconfig.yaml: |
    nameservers:
    - 2.3.4.5
    - 1.2.3.4
    searches:
    - svc.cluster.local
    - cluster.local
    options:
    - name: ndots
      value: "5"
    - name: edns0
      value: ""
```

图 3.6　configmap.yaml

```
type PodDNSConfig struct {
    Nameservers []string             `json:"nameservers,omitempty" protobuf:"bytes,1,rep,name=nameservers"`
    Searches    []string             `json:"searches,omitempty" protobuf:"bytes,2,rep,name=searches"`
    Options     []PodDNSConfigOption `json:"options,omitempty" protobuf:"bytes,3,rep,name=options"`
}

type PodDNSConfigOption struct {
    Name  string  `json:"name,omitempty" protobuf:"bytes,1,opt,name=name"`
    Value *string `json:"value,omitempty" protobuf:"bytes,2,opt,name=value"`
}
```

图 3.7　*v1.PodDNSConfig 结构

在 nameservers 内，这就是最终需要的配置文件了。它在需要的时候新生成一个 ConfigMap。

接着，Webhook 将会对端口进行监听，如果有 Pod 创建，则会开始执行修改动作，这时，Kubernetes API 服务器会向指定的服务和 URL 发送 HTTPS 请求，请求和返回的对象都是 AdmissionReview。AdmissionReview 对象描述了准入包含的内容类型，这里是 JSON，以及准入请求属性的 AdmissionRequest 对象和准入响应属性的 AdmissionResponse 对象。AdmissionRequest 对象描述了请求资源的 UID、类型、命名空间、名称等等信息；AdmissionResponse 对象描述了 UID、请求结果、执行的操作等内容。

其中一个比较重要的函数是 serve 函数，用来处理传入的 HTTP 请求，它从请求中反序列化 AdmissionReview 对象，执行一些基本的内容校验，例如判断内容类型是不是 “application/json”，再根据 URL 路径调用相应的 mutate 函数，得到对象 AdmissionResponse。然后再序列化 AdmissionReview 对象。

在 mutate 函数里，对 Pod 的 DNS 配置进行修改，并添加注解 dnsconfig-injector-webhook.scom.com/status: injected。首先将 AdmissionReview 对象反序列化，判断该 Pod 是否需要修改，这里不需要校验 kube-system 和 kube-public 两个命名空间，并且对带有注解为 “dnsconfig-injector-webhook.scom.com/inject: skip” 的 Pod 进行豁免。接着执行修改操作，将 DNS 配置挂载到/spec/dnsConfig 路径下，这里的 “/spec/dnsConfig” 路径是一个 Pod 文件描述符里 DNS 配置的路径，同时更新注解到/metadata/annotations 路径下。这样修改完后，将修改的信息序列化存入 AdmissionResponse 属性中返回。

接着，将执行 validating 函数。此函数用来验证修改的结果是否正确。这里将会对 Pod 的注解进行验证，验证其是否含有“dnsconfig-injector-webhook.scom.com/inject: skip”。同时，也会对其注解再次进行验证，判断其是否已经将 kube-dns 服务

的 ClusterIP 添加在内。

这就是大体的流程了。针对是否采用高可用 DNS，需手动给命名空间加上 label 为 dnsconfig-injector=enabled 生效，如果需要豁免，则命名空间的 label 可以为 dnsconfig-injector=disabled。这样，整个 Webhook 就基本编写完成了。将代码打包成镜像，上传至镜像仓库。这里选择了阿里云的镜像仓库。为了部署 Webhook server，需要在 Kubernetes 集群中创建一个 deployment 和 service 资源对象。deployment 内容较多，这里只选取部分重点内容进行展示，如图 3.8 所示。

```
spec:
  containers:
    - name: dnsconfig-injector
      image: registry-vpc.cn-zhangjiakou.aliyuncs.com/jstopen/dnsconfig-injector:v2
      imagePullPolicy: Always
      args:
        - -dnsCfgFile=/etc/webhook/config/dnsconfig.yaml
        - -tlsCertFile=/etc/webhook/certs/cert.pem
        - -tlsKeyFile=/etc/webhook/certs/key.pem
        - -alsologtostderr
        - -v=4
        - 2>&1
      volumeMounts:
        - name: webhook-certs
          mountPath: /etc/webhook/certs
          readOnly: true
        - name: webhook-config
          mountPath: /etc/webhook/config
  volumes:
    - name: webhook-certs
      secret:
        secretName: dnsconfig-injector-webhook-certs
    - name: webhook-config
      configMap:
        name: dnsconfig-injector-webhook-configmap
```

图 3.8　deployment 部分内容

由图 3.8 知道，image 字段的内容是刚刚上传至镜像仓库的镜像。在 args 部分，将一些配置文件加载了进去。service 内容较简单，主要是监听 443 端口，如图 3.9 所示。

将 Webhook 部署在 kube-system 命名空间下，对想要注入高可用 DNS 的命名空间加上 label，这样，该命名空间下新创建的 Pod 就会自动修改 DNS 配置了。

```yaml
apiVersion: v1
kind: Service
metadata:
  name: dnsconfig-injector-webhook-svc
  namespace: kube-system
  labels:
    app: dnsconfig-injector
spec:
  ports:
  - port: 443
    targetPort: 443
  selector:
    app: dnsconfig-injector
```

图 3.9　service.yaml

4 实验与分析

该章节描述了本文的实验方法、数据、指标、结果，以及结果分析。

4.1 测试数据

本文采用了四个具有典型代表的域名进行压力测试。分别是集群外部域名 www.baidu.com.；集群内部域名 xiaolu.testdemo.com.（通过 Ingress 暴露服务，需在 hosts 进行绑定）；同一个专有网络（Virtual Private Cloud，VPC）下的 RDS 域名 rm-k2jjl3p6mu561sv38.mysql.zhangbei.rds.aliyuncs.com. 和 Redis 域名 r-k2jfbue88fe-qivvu49.redis.zhangbei.rds.aliyuncs.com.，用于尽可能的模仿电商场景下的应用程序。

同时，本文编写了一个用于测试的应用程序，程序使用 Java 语言完成，并将该程序部署在 Kubernetes 集群内，集群内含有多个节点，节点的操作系统采用的是 centos7，并且都和 Redis、RDS 在一个 VPC 内，节点资源充足。应用容器的镜像基于 Java8，并根据本文测试需要在此基础上安装了 Jmeter，由此打包成一个新的镜像，再将应用程序的 jar 包部署上去。为了尽可能避免其他因素对 DNS 解析的影响，Java 应用程序的启动参数设置了-Dsun.net.inetaddr.ttl=-1，避免了 Java 程序自身的缓存。应用程序的 Pod 数可能有一个或多个，每个 Pod 的资源根据测试条件定；CoreDNS 副本数只有一个，本文未对其资源做任何限制，由 Kubernetes 控制。同样，对于 Node Local DNS，本文也未对其做任何资源限制。

4.2 测试方法

本文采用 Jmeter 作为测试工具进行实验。Jmeter 以线程的方式运行，每次运行一个测试计划，也叫脚本，在此层面可以决定是否独立运行每个线程组，也就是说，是多个线程组同时测试还是依次进行。一个测试计划内可以包含多个线程组，每个线程组内可以决定线程数量、延迟时间、循环次数等指标，当然也可以添加请求、各类结果分析报告等。Jmeter 的操作方式分为 GUI 模式和非 GUI 模式，非 GUI 模式对负载机的资源消耗会更小一些，本文采用非 GUI 模式进行压测。论文采用一个脚本，也是一个测试计划，里面包含了被测的机器、本文需要压测的 DNS 服务器域名、线程数量、循环次数、采样器、监听器等要素，它在压测的同

时会将包括请求结果在内的压测数据写入文件，用于后续分析。

为了尽可能的压测出 DNS 解析的真实数据，本文的测试均在应用程序所在的容器内进行，对 DNS 域名进行压测，而非通过前端请求后端服务，以避免由于网络、数据库读写耗时、程序自身缓存等因素影响 DNS 解析测试数据。同时观察应用容器、CoreDNS、Node Local DNS（如有）、Jmeter 的指标数据。

本文将通过 CoreDNS 和 Node Local DNS 的性能对比、Node Local DNS 和高可用 DNS 的可用性对比两个实验场景来描述。

4.2.1 性能对比

性能对比实验针对 CoreDNS 和 Node Local DNS。针对高可用 DNS，由于其主要解决的是现有 Node Local DNS 存在的问题，并且本文并未改变其本地缓存的特性，在 Node Local DNS 的 Pod 可用时，该方案理论上与 Node Local DNS 的性能相同，因此本文将不再对其进行类似 CoreDNS 的压力测试。

通过首先固定应用容器的 CPU 核数，来不断增加 Jmeter 的测试压力。针对四个测试域名，使用一个测试计划、四个线程组的方式，每个线程组分别采用 100 个线程数并行访问，循环次数由 100 次开始不断递增，每个线程组串行测试域名。针对每次测试，记录下当前应用容器的 CPU 使用量，当使用量超过限制核数的 90%时，增加应用容器的 CPU 数，再次重复上述测试过程，以测到该条件限制下 CoreDNS 能达到的最大的 QPS。另外，考虑到 Jmeter 本身可能的性能影响，当施加压力不断上升但应用容器的 CPU 使用量无明显上升时，将应用容器拆分成多个，使得其 CPU 总核数达到测试标准，在多个应用容器内同时进行压测。并且，又考虑到拆分多个容器可能带来未知因素的影响，论文会测试相同 CPU 核数限制下多个容器和单个容器的结果，避免无关变量的影响。也就是说，在不限制应用容器和 CoreDNS 的 CPU 核数的情况下，测出 CoreDNS 能承受的最大性能。这里需要注意的是，应用容器的 CPU 核数是手动扩容，而 CoreDNS 的核数由 Kubernetes 自动扩缩容，未对其资源进行任何限制，并且集群资源充足。这里未提及到内存，是因为在测试的过程中，始终保持应用容器的内存数值是 CPU 数值的两倍，举个例子，当应用容器的 CPU 是 2 核时，内存是 4G。这里不必纠结内存的具体数值，是按照常见的搭配取的。

针对 Node Local DNS 的测试，以现阶段电商场景下的一般需求为参考，QPS 达到 10 万即为满足需求。

4.2.2 可用性对比

可用性对比针对 Node Local DNS 及高可用 DNS。为对比 Node Local DNS 与本文提出的高可用 DNS 的不同，本文将模拟应用容器所在的节点下的 Node Local DNS 无法使用的场景，手动删除 Node Local DNS 的 Pod，测试在其删除并重新创建、正常运行这段时间内 DNS 请求的解析结果。同时，为了尽可能准确地描述 DNS 解析失败的时间，本文测试了在集群资源充足的情况下，手动删除 Node Local DNS Pod，从删除到重新运行这个过程具体时间。采用脚本不停地去使用 nslookup 命令访问各个域名，记录每次访问开始和结束的时间。按照预期，当缓存不可用时，此时的响应时间应该要比正常情况下要长，当缓存重新恢复时，响应时间也会缩短。对各个域名分别重复三次测试，记录不可用时的平均时间。

4.3 评价指标

本文实验分两组，一组是 CoreDNS 和 Node Local DNS，一组是 Node Local DNS 和高可用 DNS。针对 CoreDNS 和 Node Local DNS，本文采用相同的性能指标，包括 QPS、平均响应时间、99 分位响应时间、错误率，以及对应的样本数、应用容器使用的 CPU 核数、CoreDNS 使用的 CPU 核数。其中，CPU 使用量取测试过程中出现的最大值，判定解析出错的其中一个指标是解析时间超过 1 秒。对于有拆分容器测试的情况，则对各容器测得的 QPS、CPU 核数进行累加，平均响应时间、99 分位响应时间、错误率则是将各容器的测试数据进行统一后再计算。这里并未将内存使用量考虑在内，原因在于压测过程并不占用内存，而且经过多次实验发现，CoreDNS 内存使用量基本小于 100M，应用程序使用内存量基本小于 2G，并且应用程序使用内存量较大主要是因为 Jmeter 在测试过程中需要将每一次的测试结果写入文件，因此内存这一项忽略不计。本文认为，在错误率不超过 0.01%的条件下，CoreDNS 能承受的 QPS 越高，则性能越好。

针对 Node Local DNS 和高可用 DNS，本文采用 Node Local DNS 不可用场景下 DNS 请求解析结果作为评价指标。特别的是，需要关注各自在 Node Local DNS 不可用时 DNS 请求的时间，这对于评价本文的贡献十分重要。

4.4 实验结果

4.4.1 可用性对比

为了对比 Node Local DNS 和本文提出的高可用 DNS 之间的差别，将手动删除应用容器所在节点上的 Node Local DNS，同时使用 Jmeter 进行测试，观察在 Node Local DNS 重新恢复正常之前的 DNS 解析结果。不出意料的是，在这段时间内，DNS 解析结果全部出错，Jmeter 返回错误码 500，错误信息 java.net.PortUnreachableException。

需要注意的是，本文采用 Jmeter 进行压测时，由于是直接对 DNS 服务器进行压测，因此指定了服务器的 IP，也就是说，截至到目前为止的测试，CoreDNS 的是对指向 CoreDNS 的 kube-dns 服务的 IP 进行压测，Node Local DNS 的是对指向 DaemonSet 容器的 IP 进行压测。可以发现之前对于 Node Local DNS 的压测是存在漏洞的，也就是说，在 Node Local DNS 已经不可用的情况下，再去请求该服务器，得到的结果一定是解析失败。因此，不应该指定 DNS 服务器的 IP。在这里简单地进行测试。正常情况下，进入容器执行命令 nslookup，nslookup 会发送命令给对应的 DNS 服务器，就可以获取到域名的 IP 地址以及 DNS 服务器的 IP 地址。现在再次手动删除 Node Local DNS，然后在应用容器内执行 nslookup 命令，结果发现相比未删除 Node Local DNS 时，出现了较长时间的无响应状态才得到解析结果，此时新的 Node Local DNS 已经正常运行。从中也可以推断出，如果 Node Local DNS 因为其他原因，例如拉取镜像失败、集群资源不足等情况一直无法正常运行时，DNS 解析请求会一直无响应。

Node Local DNS 和高可用 DNS 测试的不可运行时间如表 4.1 所示。

表 4.1 可用性测试结果

域名	Node Local DNS 平均不可用时间（毫秒）	高可用 DNS 平均不可用时间（毫秒）
www.baidu.com.	5010	2009
xiaolu.testdemo.com.	5011	2010
rm-k2jjl3p6mu561sv38.mysql.zhangbei.rds.aliyuncs.com.	5011	2010
r-k2jfbue88feqivvu49.redis.zhangbei.rds.aliyuncs.com.	10012	2008

可以看到，除了 Redis 的域名解析时间在 10 秒左右，其他都在 5 秒左右，关于 Redis 的域名解析时间较其他较长不得而知，但又观察了在这个测试过程中各

个域名在缓存正常时候的解析时间，Redis 在十几毫秒左右，百度域名在 1 毫秒左右，相比不可用时的解析时间可以忽略不计。再次强调，测试时是手动删除了缓存，由于集群资源充足且没有其他故障，缓存可以正常重建；而在实际生产中，不会去手动删除 Pod，通常都是由一些故障导致 Pod 不可用，而这个修复时间不可预估，且可能会远大于这里测试的不可用时间。

至此，本文认为，Node Local DNS 的可用性还有待提升。

针对高可用 DNS，在 Node Local DNS 的 Pod 不可用时的 DNS 响应时间约 2 秒左右，由 CoreDNS 服务器解析。相比起前一节测试的 Node Local DNS 不可用时间为 5 ~ 10 秒看起来似乎差别不大。但需要注意的是，这里 DNS 服务器一直处于正常的状态，缓存不可用时使用的 CoreDNS 解析；而前一节里 DNS 服务器处于不可用的状态。两者在这个地方有本质差异。

接着，本文将分析高可用 DNS 平均响应时间在 2 秒的原因。因为在 CoreDNS 的测试结果里可以看到，平均响应时间不超过 10 毫秒。同样是使用 CoreDNS 服务器解析，为什么响应时间差别这么大呢？起初怀本文疑是/etc/resolv.conf 文件里 options 属性的设置问题，这里设置了“timeout:3 attempts:1”，这意味着每次请求超时时间是 3 秒，重试 1 次。怀疑是第一次请求缓存时，请求超时 1 秒，重试，再次请求超时 1 秒，这样就是 2 秒，接着再请求 CoreDNS，请求时间不超过 10 毫秒，这和测试结果一致。为了验证这种猜想，重复实验，多次修改 timeout 和 attempts 的值，但是响应时间始终是 2 秒左右，因此否定了是 options 选项的原因。

需要注意的是，这里并非是缓存可以连接，只是响应超时导致重试，这样才会根据 options 选项超时重试。这里是缓存不可用，因此实际上它并不会根据 options 的设置来。于是，本文决定对其进行抓包分析。手动删除缓存，同时，在该缓存所在节点上的应用容器里使用 nslookup 对域名进行请求，并且使用 tcpdump 命令进行抓包分析，具体的命令是“tcpdump -nn -i eth0 icmp or udp port 53”，抓包结果如图 4.1 所示。

```
14:16:55.917353 IP 172.20.0.199.39214 > 169.254.20.10.53: 45565+ A? r-k2jfbue88feqivvu49.redis.zhangbei.rds.aliyuncs.com. (70)
14:16:55.917377 IP 169.254.20.10 > 172.20.0.199: ICMP 169.254.20.10 udp port 53 unreachable, length 106
14:16:56.917236 IP 172.20.0.199.42381 > 172.21.0.10.53: 45565+ A? r-k2jfbue88feqivvu49.redis.zhangbei.rds.aliyuncs.com. (70)
14:16:56.917987 IP 172.21.0.10.53 > 172.20.0.199.42381: 45565* 1/0/0 A 192.168.17.125 (138)
14:16:56.918283 IP 172.20.0.199.57825 > 169.254.20.10.53: 45355+ AAAA? r-k2jfbue88feqivvu49.redis.zhangbei.rds.aliyuncs.com. (70)
14:16:56.918301 IP 169.254.20.10 > 172.20.0.199: ICMP 169.254.20.10 udp port 53 unreachable, length 106
14:16:57.918402 IP 172.20.0.199.46820 > 172.21.0.10.53: 45355+ AAAA? r-k2jfbue88feqivvu49.redis.zhangbei.rds.aliyuncs.com. (70)
14:16:57.919230 IP 172.21.0.10.53 > 172.20.0.199.46820: 45355* 0/1/0 (171)
```

图 4.1　缓存不可用时的抓包结果

可以看到，2 秒的时间几乎都消耗在了查询 A 记录和 AAAA 记录上。在之前

使用 Jmeter 的压测中，都是查询的 A 记录，因此，如果不使用 ipv6，也就是不查询 AAAA 记录，则延迟时间可以再减少 1 秒。接着，又对正常情况下的 DNS 请求进行了抓包，抓包结果如图 4.2，时间消耗在 3 毫秒左右。

```
14:28:39.524704 IP 172.20.0.199.54197 > 169.254.20.10.53: 3841+ A? r-k2jfbue88feqivvu49.redis.zhangbei.rds.aliyuncs.com. (70)
14:28:39.526593 IP 169.254.20.10.53 > 172.20.0.199.54197: 3841* 1/0/0 A 192.168.17.125 (138)
14:28:39.526837 IP 172.20.0.199.56314 > 169.254.20.10.53: 3021+ AAAA? r-k2jfbue88feqivvu49.redis.zhangbei.rds.aliyuncs.com. (70)
14:28:39.527486 IP 169.254.20.10.53 > 172.20.0.199.56314: 3021* 0/1/0 (171)
```

图 4.2　缓存可用时的抓包结果

尽管如此，比前一小节中 Node Local DNS 的响应情况要好很多。

4.4.2　性能对比

表格 4.2 是 CoreDNS 的测试数据。注，样本数指测试的四个域名的总样本数，例如测试百度域名为 100 线程数循环 3000 次，则百度域名样本数为 30 万次，四个域名总样本数为 120 万次；应用容器 CPU 使用量指一个或多个（如有）应用容器使用的总 CPU 核数。

表 4.2　CoreDNS 测试结果

样本数（万）	QPS	平均响应时间（毫秒）	99 分位响应时间（毫秒）	错误率（%）	应用容器 CPU 使用量（核）	CoreDNS CPU 使用量（核）
20	7884	5	60	0.00	0.99	0.41
120	13608	5	74	0.00	1.99	0.94
160	20892	4	62	0.00	2.97	1.43
200	28426	3	44	0.00	3.97	1.41
200	35897	2	25	0.00	4.99	2.39
400	43134	2	15	0.00	5.94	2.76
600	43850	2	16	0.00	6.97	2.85
600	46781	1	24	0.00	7.99	2.87
600	51849	1	16	0.00	8.89	3.08
800	56804	1	15	0.00	9.37	3.31

从表格 4.2 可以看到，随着样本数的增加，QPS、应用容器 CPU 使用量、CoreDNS 的 CPU 使用量都在随之增加。到应用容器 CPU 使用量接近 10 核时，CoreDNS 的 CPU 使用量接近 4 核，QPS 约为 5.6 万。但值得注意的是，在实验过程中，尝试增加应用容器的 CPU 核数、将单个应用容器拆分成多个应用容器并保持总 CPU 核数不变、增加样本数、重复测试，但无一例外的是，应用容器的 CPU 使用量维持在接近 10 核，QPS 在 5.6 万左右，CoreDNS 的 CPU 使用量在 3 ~ 4 核。

本文猜想，在 DNS 解析过程中，由于 CoreDNS 和应用容器可能不在同一节点，需要经历跨节点请求，如本文 2.1.4 小节所述，此过程中存在多次解包和封包，对性能有一定影响，限制了 CoreDNS 的性能，但具体影响有多大，本文中未做分析，也不在本文重点讨论范围内。

因此得出结论，在集群单纯使用 CoreDNS 的情况下，DNS 解析最高可达的 QPS 约为 5.6 万，考虑到电商场景下 10 万左右的 QPS 访问，这一结果远不足以支撑现阶段的需求。

现在在集群中安装 Node Local DNS，并重新部署应用容器，其他条件不变。测量指标相比 CoreDNS 多了一项 Node Local DNS 的 CPU 使用量，具体压测结果见表 4.3。

表 4.3　Node Local DNS 测试结果

样本数（万）	QPS	平均响应时间（毫秒）	99 分位响应时间（毫秒）	错误率（%）	应用容器 CPU 使用量（核）	CoreDNS CPU 使用量（核）	Node Local DNS CPU 使用量（核）
800	45963	2	18	0.00	6.087	<0.1	4.16
800	70621	3	21	0.00	10.051	<0.1	7.56
800	86237	2	19	0.00	11.977	<0.1	10.90
800	106364	3	16	0.00	15.56	<0.1	13.13

由表 4.3 中可以看出，随着应用容器 CPU 使用量上升，QPS 也随之上升，未出现 CoreDNS 的性能瓶颈。当应用容器 CPU 核数达到约 16 核时，QPS 达到 10 万，满足现阶段的需求，因此不再继续测试。理论上，Node Local DNS 的性能不应该存在上限，除非在某一时刻极大量未缓存过的 DNS 域名解析请求到 CoreDNS，或集群资源达到上限，应用容器或 Node Local DNS 无法再承受巨大的流量。此外，可以发现，CoreDNS 的 CPU 使用量一直低于 0.1 核，这是因为 DNS 解析结果都缓存在了 Node Local DNS，无需再通过 CoreDNS 进行解析。

因此，可以得出结论，Node Local DNS 的性能要比 CoreDNS 更好，对于现阶段 10 万 QPS 请求的需求已经能够满足。

4.5　误差分析

为了更好地理解本题所提出的高可用 DNS 工具的一些限制，本文将从以下几点阐述。

1. 本文使用 Jmeter 工具进行压测和数据分析，该工具的测试结果并不完全准确。例如，Jmeter 会进行多线程的创建、回收，将单次测试结果写入文件，等等，这些过程都需要占用资源，可能造成测得的 QPS 比 DNS 服务器真实能承受的值偏低。
2. 尽管本文已尽可能的减少非 Node Local DNS 缓存带来的影响，比如设置 Java 的启动参数禁止缓存，但仍然可能存在操作系统等层面的缓存，对实验数据造成干扰。此外，CoreDNS 存在默认的 30 秒缓存，相比于每隔 30 秒以上请求一次，连续压测的平均响应时间会更短一些，但这也更符合实际情况。生产环境下 QPS 够大才会出现 CoreDNS 解析失败的问题，才需要使用 Node Local DNS。
3. 本文是直接压测的 DNS 服务器，而并非真实模拟客户操作从前端进行请求，后端进行诸如数据库查询等操作后返回结果。这中间少了网络传输、数据库处理等众多消耗资源的过程。并且，正常情况下，Java 应用程序本身就带有缓存，而 PHP 应用程序并不具备很好的缓存机制，也会造成和真实情况有偏差。此外，在压测过程中请求失败时，线程不会重试，而在实际用户操作过程中，面对请求失败或无响应的情况下，可能会反复重试刷新，并且，客户端、服务端线程可能都会进行重试，从而造成请求堆积，失败越多请求越多，对服务的负载过大，进而导致内存使用过高，产生 OOM（内存不足），造成 Pod 重启。但考虑到本文的目的是高可用 DNS 的设计，压测 DNS 服务器更加准确。
4. 本文只使用了四个测试域名，无论是否有缓存，这四个域名都不会对 CoreDNS 造成解析压力，也就是说，本文并未考虑过多的各不相同的域名对 CoreDNS 带来的影响。此外，在 2.2 小节提到 DNS 配置文件内有三个属性，其中有一个属性是 options，里面会有 ndots 这个配置，在实验中，ndots 选择的是 5，也就说是，在测试外部域名和内部域名的时候，其实存在多次请求浪费的情况，会造成一定的解析延迟，但 RDS 和 Redis 不会造成请求浪费，一次就可以解析成功。这一点对于 CoreDNS、Node Local DNS、高可用 DNS 都会有影响，对于 CoreDNS 可能影响会更大一些。但本文的目的是测试缓存不可用时的 DNS 解析能力，所以也可以忽略此种情况。

5 结语

5.1 结论

本文提出了一个高可用的 DNS 工具，用于解决现有 Node Local DNS 存在的可用性不足的问题。它通过 MutatingAdmissionWebhook 实现，在 Pod 新创建时修改 DNS 配置，并对已修改过的 Pod 打上注解，以此达到在 Node Local DNS 不可用时 DNS 请求依然能解析成功的目的。更详细地说，它为 DNS Config 的 nameservers 字段新增了服务 kube-dns 的 IP，让 CoreDNS 成为 DNS 解析服务器的备选项。

在实验过程中，选择了四个测试域名对 CoreDNS、Node Local DNS、高可用 DNS 分别进行了测试。整个实验大体可以分为两部分，一部分是 CoreDNS 和 Node Local DNS 的性能测试；另一部分是 Node Local DNS 和高可用 DNS 的可用性测试。实验中得出结论，Node Local DNS 可以承受 10 万 QPS 的需求，而 CoreDNS 最多能承受约 5 万的 QPS；在 DNS 缓存不可用时，Node Local DNS 的 DNS 请求均失败，而高可用 DNS 的请求可以正常解析。

因此，本文提出的高可用 DNS 满足可用性要求，并且性能并未受影响。

5.2 问题与展望

本文的研究还存在一些不足，可以在今后的工作中进一步改进：

1. 本文只使用了少量的域名进行测试，当有大量未缓存过的域名同时请求 DNS 服务器时，可能会出现响应延迟的情况。因为在高可用 DNS 中，CoreDNS 作为备用的服务器，只有当缓存不可用或者域名未缓存时才会请求到 CoreDNS，并非直接请求到 CoreDNS，所以中间会存在延时。并且，CoreDNS 解析能力有上限，在 DNS 请求过多时，即使是采用了高可用 DNS，也可能会出现解析失败的情况。因此，对于这种情况，还需要继续研究，找到更好的解决方案。
2. 本文的高可用 DNS 仍会出现 2 秒的响应延迟，这对于生产环境来说仍然还需要改进。如何让这个时间降到最低，是今后的工作方向。
3. 本文只针对 DNS 解析这一部分做了改进，但从客户请求到响应这整个过程并未做实验与优化，因此可能还存在其他方面需要改进，这也和本文最初的目标是一致的，给用户带来良好的使用体验，保障系统的稳定运行。

参考文献

[1] 中国企业云服务行业研究报告 2016 年 [C]//上海艾瑞市场咨询有限公司. 艾瑞咨询系列研究报告（2016 年第 11 期）. 上海艾瑞市场咨询有限公司, 2016: 69.

[2] 张斌. Kubernetes 容器集群下分布式事务解决方案的研究 [D]. 浙江理工大学, 2020.

[3] HäKLI A, TAIBI D, SYSTA K. Towards cloud native continuous delivery: An industrial experience report[C]//2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). 2018: 314-320.

[4] 单朋荣. 面向应用的容器集群弹性伸缩方法的设计与实现 [D]. 齐鲁工业大学, 2020.

[5] 中国容器云市场研究报告 艾瑞云原生系列报告（一）2020 年 [C]//上海艾瑞市场咨询有限公司. 艾瑞咨询系列研究报告（2020 年第 12 期）. 上海艾瑞市场咨询有限公司, 2020: 56.

[6] 武志学. 云计算虚拟化技术的发展与趋势 [J]. 计算机应用, 2017, 37(04): 915-923.

[7] 胡晓亮. 基于 Kubernetes 的容器云平台设计与实现 [D]. 西安电子科技大学, 2019.

[8] 李翔. 在私有 Kubernetes 集群中实现服务的负载均衡 [J]. 电子技术与软件工程, 2020(04): 36-38.

[9] KAPOčIUS N. Performance studies of kubernetes network solutions[C]//2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream). 2020: 1-6.

[10] BHARDWAJ S, JAIN L, JAIN S. Cloud computing: A study of infrastructure as a service (iaas)[J]. International Journal of engineering and information Technology, 2010, 2(1): 60-63.

[11] NASHAAT H, ASHRY N, RIZK R. Smart elastic scheduling algorithm for virtual machine migration in cloud computing[J]. The Journal of Supercomputing, 2019, 75(7): 3842-3865.

[12] DUTTA A, PENG G C A, CHOUDHARY A. Risks in enterprise cloud computing:

the perspective of it experts[J]. Journal of Computer Information Systems, 2013, 53 (4): 39-48.

[13] 张悦. 企业弹性应用云迁移的资源分配优化模型与算法研究 [D]. 燕山大学, 2020.

[14] 周佳威. Kubernetes 跨集群管理的设计与实现 [D]. 浙江大学, 2017.

[15] 陈文楷. 基于 docker 容器的高并发 web 系统架构设计与实现 [D]. 北京邮电大学, 2019.

[16] Using nodelocal dnscache in kubernetes clusters[EB/OL]. 2020[2000-11-09]. https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/.

[17] 刘渊, 乔巍. 云环境下基于 Kubernetes 集群系统的容器网络研究与优化 [J]. 信息网络安全, 2020, 20(3): 36.

[18] XIE B, SUN G, MA G. Docker based overlay network performance evaluation in large scale streaming system[C]//2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). IEEE, 2016: 366-369.

[19] Using admission controllers[EB/OL]. 2021[2021-04-30]. https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/.

致谢

东湖之滨，珞珈山上。

转眼四年即逝，当初入学时的场景仍历历在目。怀揣着希望和热爱，这几年过得充实而美好。春看樱花绽放，游人如织；夏听蝉声一片，绿荫乘凉；秋见银杏泛黄，夕阳西下；冬盼大雪纷飞，银装素裹。武大之美，不仅在于自然风光，更与其人文情怀息息相关。丰富多样的课程种类，实力雄厚的师资队伍，令人眼花缭乱的社团活动，实在丰厚的奖助学金……这一切的一切似乎都在传达，自由包容如武大，不论心之所向，她刚刚 © 将永远提供最大的支持。一直记得有位老师曾在课堂上说，年轻人要多做“无用之事”。于是这四年来，听课、科研、比赛、活动、支教、旅游、实习、兼职、恋爱……一起拼凑成了我这五彩斑斓的生活，做了太多从未想过的事，踏出了太多的第一步。樱顶老图、梅操、东湖、万林博物馆、教五草坪、工菜、信操、青楼、图书馆……一个个不能再熟悉的地点，都留下了太多的故事，有泪有笑，有生气也有感动。也是这些经历，将迷茫的我逐渐指引到了一条未来更适合我的路，工作。自强、弘毅、求是、拓新，这八个字定将牢记于心，感恩武大，祝愿将来更好。

这一路上，认识了很多德才兼备的老师，他们的言谈举止对我的学习成长、思想行动都有着非同小可的影响，“听君一席话，胜读十年书”，从此打开了一扇又一扇新的大门。遇到过无数优秀且努力的同学，他们让我在求学路上更加勤勉、不敢懈怠，也让我明白，评判优秀的指标不再单一，每个人都有自己的舞台。结识了各类靠谱且认真的队友，让我们的团队活动可以更加顺利地进行，也让我发现了人生的多种可能，多种色彩。交了一些真诚有趣的朋友，成长路上总有人相伴，失败、胆怯、让人崩溃焦虑的日子，变得不再难熬；收获、勇敢、让人激动开心的时刻，也有人分享见证。接触了经验丰富、待人友好的公司前辈们，跟着他们让我的专业能力更加扎实，职业规划更加明晰。人海茫茫，相遇即缘分，感恩遇到的人，山水一程，三生有幸。

这篇论文，倾注了贾向阳导师、公司团队前辈们的诸多心血，在此感恩不尽。曾听过贾向阳导师的课，对导师的专业、负责、认真、耐心留下了非常深刻的印象。这次即便是在校外完成毕设，隔着屏幕依旧能感受到导师的用心，导师对论文的框架、方向、内容等都提出了非常多的建设性的意见，保持紧密联系也使得

论文时间节点安排都在预期之内。选题由团队前辈涂靖给出，于我而言颇具难度，一直到定稿这个过程中间的技术学习、敲定方向、开发实验、论文开题定稿，都离不开团队前辈们一步一步的耐心指导。每每有疑惑、遇到瓶颈时，都能得到专业细致的解答，对论文的推进起了非常大的作用。此外，感谢团队的包容与开放、理解与支持，让我在这个过程中收获颇丰。

最后，感谢明智的父母，对我的一次次选择都给予了支持。未来路上，将秉持着一份善良、正直、勇敢、热爱，继续前行。希望多年后，归来仍是珞珈一少年。