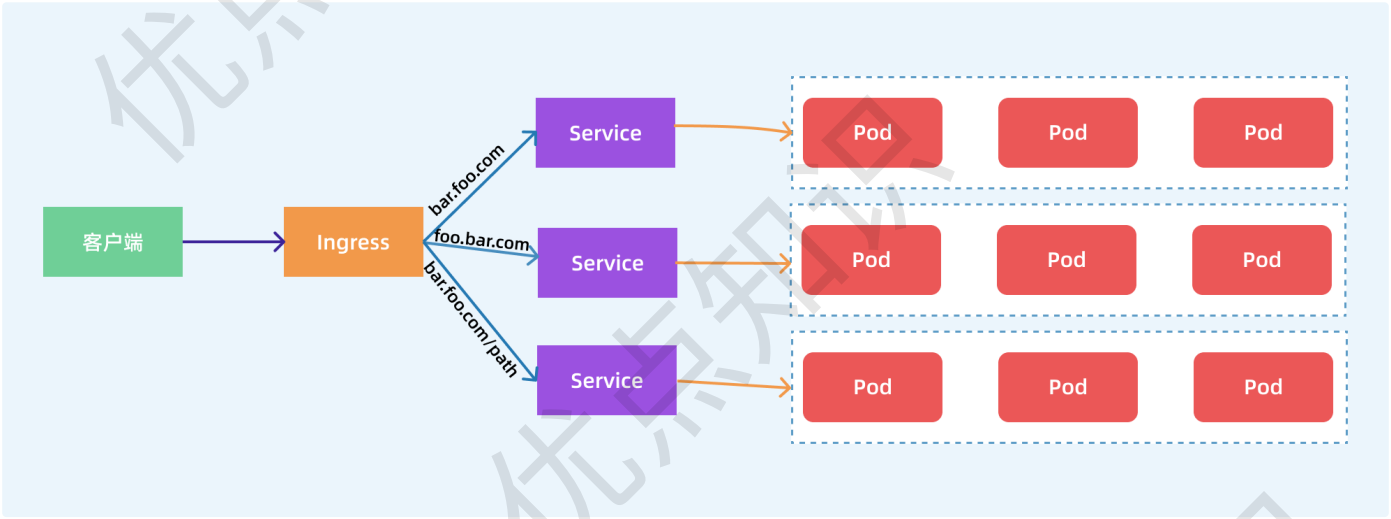
Ingress

前面我们学习了在 Kubernetes 集群内部使用 kube-dns 实现服务发现的功能，那么我们部署在 Kubernetes 集群中的应用如何暴露给外部的用户使用呢？我们知道可以使用 NodePort 和 LoadBlancer 类型的 Service 可以把应用暴露给外部用户使用，除此之外，Kubernetes 还为我们提供了一个非常重要的资源对象可以用来暴露服务给外部用户，那就是 Ingress 。对于小规模的应用我们使用 NodePort 或许能够满足我们的需求，但是当你的应用越来越多的时候，你就会发现对于 NodePort 的管理就非常麻烦了，这个时候使用 Ingress 就非常方便了，可以避免管理大量的端口。

资源对象

Ingress 资源对象是 Kubernetes 内置定义的一个对象，是从 Kubernetes 集群外部访问集群的一个入口，将外部的请求转发到集群内不同的 Service 上，其实就相当于 nginx、haproxy 等负载均衡代理服务器，可能你会觉得我们直接使用 nginx 就实现了，但是只使用 nginx 这种方式有很大缺陷，每次有新服务加入的时候怎么改 Nginx 配置？不可能让我们去手动更改或者滚动更新前端的 Nginx Pod 吧？那我们再加上一个服务发现的工具比如 consul 如何？貌似是可以，对吧？Ingress 实际上就是这样实现的，只是服务发现的功能自己实现了，不需要使用第三方的服务了，然后再加上一个域名规则定义，路由信息的刷新依靠 Ingress Controller 来提供。



Ingress Controller 可以理解为一个监听器，通过不断地监听 kube-apiserver，实时的感知后端 Service、Pod 的变化，当得到这些信息变化后，Ingress Controller 再结合 Ingress 的配置，更新反向代理负载均衡器，达到服务发现的作用。其实这点和服务发现工具 consul、 consul-template 非常类似。

定义

一个常见的 Ingress 资源清单如下所示：

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: demo-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - http:
        paths:
```

```
          - path: /testpath
            pathType: Prefix
            backend:
              service:
                name: test
                port:
                  number: 80
```

上面这个 Ingress 资源的定义，配置了一个路径为 /testpath 的路由，所有 /testpath/** 的入站请求，会被 Ingress 转发至名为 test 的服务的 80 端口的 / 路径下。可以将 Ingress 狭义的理解为 Nginx 中的配置文件 nginx.conf 。

此外 Ingress 经常使用注解 annotations 来配置一些选项，当然这具体取决于 Ingress 控制器的实现方式，不同的 Ingress 控制器支持不同的注解。

另外需要注意的是当前集群版本是 v1.25，这里使用的 apiVersion 是 networking.k8s.io/v1，所以如果是之前版本的 Ingress 资源对象需要进行迁移。 Ingress 资源清单的描述我们可以使用 kubectl explain 命令来了解：

```
➜ kubectl explain ingress.spec
KIND:     Ingress
VERSION:  networking.k8s.io/v1

RESOURCE: spec <Object>

DESCRIPTION:
     Spec is the desired state of the Ingress. More info:
     https://git.k8s.io/community/contributors/devel/sig-architecture/api-
conventions.md#spec-and-status

     IngressSpec describes the Ingress the user wishes to exist.

FIELDS:
   defaultBackend       <Object>
     DefaultBackend is the backend that should handle requests that don't match
     any rule. If Rules are not specified, DefaultBackend must be specified. If
     DefaultBackend is not set, the handling of requests that do not match any
     of the rules will be up to the Ingress controller.

   ingressClassName     <string>
     IngressClassName is the name of the IngressClass cluster resource. The
     associated IngressClass defines which controller will implement the
     resource. This replaces the deprecated `kubernetes.io/ingress.class`
     annotation. For backwards compatibility, when that annotation is set, it
     must be given precedence over this field. The controller may emit a warning
     if the field and annotation have different values. Implementations of this
     API should ignore Ingresses without a class specified. An IngressClass
     resource may be marked as default, which can be used to set a default value
     for this field. For more information, refer to the IngressClass
     documentation.

   rules        <[]Object>
```

```
    A list of host rules used to configure the Ingress. If unspecified, or no
    rule matches, all traffic is sent to the default backend.

  tls  <[]Object>
    TLS configuration. Currently the Ingress only supports a single TLS port,
    443. If multiple members of this list specify different hosts, they will be
    multiplexed on the same port according to the hostname specified through
    the SNI TLS extension, if the ingress controller fulfilling the ingress
    supports SNI.
```

从上面描述可以看出 Ingress 资源对象中有几个重要的属性：defaultBackend、ingressClassName、rules、tls。

rules

其中核心部分是 rules 属性的配置，每个路由规则都在下面进行配置：

- host：可选字段，上面我们没有指定 host 属性，所以该规则适用于通过指定 IP 地址的所有入站 HTTP 通信，如果提供了 host 域名，则 rules 则会匹配该域名的相关请求，此外 host 主机名可以是精确匹配（例如 foo.bar.com）或者使用通配符来匹配（例如 *.foo.com）。
- http.paths：定义访问的路径列表，比如上面定义的 /testpath，每个路径都有一个由 backend.service.name 和 backend.service.port.number 定义关联的 Service 后端，在控制器将流量路由到引用的服务之前，host 和 path 都必须匹配传入的请求才行。
- backend：该字段其实就是用来定义后端的 Service 服务的，与路由规则中 host 和 path 匹配的流量会将发送到对应的 backend 后端去。

> 此外一般情况下在 Ingress 控制器中会配置一个 defaultBackend 默认后端，当请求不匹配任何 Ingress 中的路由规则的时候会使用该后端。defaultBackend 通常是 Ingress 控制器的配置选项，而非在 Ingress 资源中指定。

Resource

backend 后端除了可以引用一个 Service 服务之外，还可以通过一个 resource 资源进行关联，Resource 是当前 Ingress 对象命名空间下引用的另外一个 Kubernetes 资源对象，但是需要注意的是 Resource 与 Service 配置是互斥的，只能配置一个，Resource 后端的一种常见用法是将所有入站数据导向带有静态资产的对象存储后端，如下所示：

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource-backend
spec:
  rules:
    - http:
        paths:
          - path: /icons
            pathType: ImplementationSpecific
            backend:
              resource:
                apiGroup: k8s.example.com
```

```
            kind: StorageBucket
            name: icon-assets
```

该 Ingress 资源对象描述了所有的 `/icons` 请求会被路由到同命名空间下的名为 `icon-assets` 的 `StorageBucket` 资源中去进行处理。

pathType

上面的示例中在定义路径规则的时候都指定了一个 `pathType` 的字段，事实上每个路径都需要有对应的路径类型，当前支持的路径类型有三种：

- `ImplementationSpecific`：该路径类型的匹配方法取决于 `IngressClass`，具体实现可以将其作为单独的 pathType 处理或者与 `Prefix` 或 `Exact` 类型作相同处理。
- `Exact`：精确匹配 URL 路径，且区分大小写。
- `Prefix`：基于以 `/` 分隔的 URL 路径前缀匹配，匹配区分大小写，并且对路径中的元素逐个完成，路径元素指的是由 `/` 分隔符分隔的路径中的标签列表。

`Exact` 比较简单，就是需要精确匹配 URL 路径，对于 `Prefix` 前缀匹配，需要注意如果路径的最后一个元素是请求路径中最后一个元素的子字符串，则不会匹配，例如 `/foo/bar` 可以匹配 `/foo/bar/baz`，但不匹配 `/foo/barbaz`，可以查看下表了解更多的匹配场景（来自官网）：

类型	路径	请求路径	匹配与否?
Prefix	`/`	（所有路径）	是
Exact	`/foo`	`/foo`	是
Exact	`/foo`	`/bar`	否
Exact	`/foo`	`/foo/`	否
Exact	`/foo/`	`/foo`	否
Prefix	`/foo`	`/foo`, `/foo/`	是
Prefix	`/foo/`	`/foo`, `/foo/`	是
Prefix	`/aaa/bb`	`/aaa/bbb`	否
Prefix	`/aaa/bbb`	`/aaa/bbb`	是
Prefix	`/aaa/bbb/`	`/aaa/bbb`	是，忽略尾部斜线
Prefix	`/aaa/bbb`	`/aaa/bbb/`	是，匹配尾部斜线
Prefix	`/aaa/bbb`	`/aaa/bbb/ccc`	是，匹配子路径
Prefix	`/aaa/bbb`	`/aaa/bbbxyz`	否，字符串前缀不匹配
Prefix	`/`, `/aaa`	`/aaa/ccc`	是，匹配 `/aaa` 前缀
Prefix	`/`, `/aaa`, `/aaa/bbb`	`/aaa/bbb`	是，匹配 `/aaa/bbb` 前缀
Prefix	`/`, `/aaa`, `/aaa/bbb`	`/ccc`	是，匹配 `/` 前缀
Prefix	`/aaa`	`/ccc`	否，使用默认后端
混合	`/foo` (Prefix), `/foo` (Exact)	`/foo`	是，优选 Exact 类型

> 在某些情况下，Ingress 中的多条路径会匹配同一个请求，这种情况下最长的匹配路径优先，如果仍然有两条同等的匹配路径，则精确路径类型优先于前缀路径类型。

IngressClass

Kubernetes 1.18 起，正式提供了一个 IngressClass 资源，作用与 kubernetes.io/ingress.class 注解类似，因为可能在集群中有多个 Ingress 控制器，可以通过该对象来定义我们的控制器，例如：

```yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: external-lb
spec:
  controller: nginx-ingress-internal-controller
  parameters:
    apiGroup: k8s.example.com
    kind: IngressParameters
    name: external-lb
```

其中重要的属性是 `metadata.name` 和 `spec.controller`，前者是这个 `IngressClass` 的名称，需要设置在 Ingress 中，后者是 Ingress 控制器的名称。

Ingress 中的 `spec.ingressClassName` 属性就可以用来指定对应的 IngressClass，并进而由 IngressClass 关联到对应的 Ingress 控制器，如：

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp
spec:
  ingressClassName: external-lb # 上面定义的 IngressClass 对象名称
  defaultBackend:
    service:
      name: myapp
      port:
        number: 80
```

不过需要注意的是 `spec.ingressClassName` 与老版本的 `kubernetes.io/ingress.class` 注解的作用并不完全相同，因为 `ingressClassName` 字段引用的是 `IngressClass` 资源的名称，`IngressClass` 资源中除了指定了 Ingress 控制器的名称之外，还可能会通过 `spec.parameters` 属性定义一些额外的配置。

比如 `parameters` 字段有一个 `scope` 和 `namespace` 字段，可用来引用特定于命名空间的资源，对 Ingress 类进行配置。 `scope` 字段默认为 `Cluster`，表示默认是集群作用域的资源。将 `scope` 设置为 `Namespace` 并设置 `namespace` 字段就可以引用某特定命名空间中的参数资源，比如：

```yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: external-lb
spec:
  controller: nginx-ingress-internal-controller
  parameters:
    apiGroup: k8s.example.com
    kind: IngressParameters
    name: external-lb
    namespace: external-configuration
    scope: Namespace
```

由于一个集群中可能有多个 Ingress 控制器，所以我们还可以将一个特定的 IngressClass 对象标记为集群默认是 Ingress 类。只需要将一个 IngressClass 资源的 ingressclass.kubernetes.io/is-default-class 注解设置为 true 即可，这样未指定 ingressClassName 字段的 Ingress 就会使用这个默认的 IngressClass。

> 如果集群中有多个 IngressClass 被标记为默认，准入控制器将阻止创建新的未指定 ingressClassName 的 Ingress 对象。最好的方式还是确保集群中最多只能有一个 IngressClass 被标记为默认。

TLS

Ingress 资源对象还可以用来配置 Https 的服务，可以通过设定包含 TLS 私钥和证书的 Secret 来保护 Ingress。Ingress 只支持单个 TLS 端口 443，如果 Ingress 中的 TLS 配置部分指定了不同的主机，那么它们将根据通过 SNI TLS 扩展指定的主机名 （如果 Ingress 控制器支持 SNI）在同一端口上进行复用。需要注意 TLS Secret 必须包含名为 tls.crt 和 tls.key 的键名，例如：

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 编码的 cert
  tls.key: base64 编码的 key
type: kubernetes.io/tls
```

在 Ingress 中引用此 Secret 将会告诉 Ingress 控制器使用 TLS 加密从客户端到负载均衡器的通道，我们需要确保创建的 TLS Secret 创建自包含 https-example.foo.com 的公用名称的证书，如下所示：

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
    - hosts:
        - https-example.foo.com
      secretName: testsecret-tls
  rules:
    - host: https-example.foo.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: service1
                port:
                  number: 80
```

现在我们了解了如何定义 Ingress 资源对象了，但是仅创建 Ingress 资源本身没有任何效果。还需要部署 Ingress 控制器，例如 ingress-nginx，现在可以供大家使用的 Ingress 控制器有很多，比如 traefik、nginx-controller、Kubernetes Ingress Controller for Kong、HAProxy Ingress controller，当然你也可以自己实现一个 Ingress Controller，现在普遍用得较多的是 ingress-nginx、apisix 以及 traefik，traefik 的性能比 ingress-nginx 差，但是配置使用要简单许多。

> 实际上社区目前还在开发一组高配置能力的 API，被称为 Gateway API，新 API 会提供一种 Ingress 的替代方案，它的存在目的不是替代 Ingress，而是提供一种更具配置能力的新方案。

ingress-nginx

我们已经了解了 Ingress 资源对象只是一个路由请求描述配置文件，要让其真正生效还需要对应的 Ingress 控制器才行，Ingress 控制器有很多，这里我们先介绍使用最多的 ingress-nginx，它是基于 Nginx 的 Ingress 控制器。

运行原理

ingress-nginx 控制器主要是用来组装一个 nginx.conf 的配置文件，当配置文件发生任何变动的时候就需要重新加载 Nginx 来生效，但是并不会只在影响 upstream 配置的变更后就重新加载 Nginx，控制器内部会使用一个 lua-nginx-module 来实现该功能。

我们知道 Kubernetes 控制器使用控制循环模式来检查控制器中所需的状态是否已更新或是否需要变更，所以 ingress-nginx 需要使用集群中的不同对象来构建模型，比如 Ingress、Service、Endpoints、Secret、ConfigMap 等可以生成反映集群状态的配置文件的对象，控制器需要一直 Watch 这些资源对象的变化，但是并没有办法知道特定的更改是否会影响到最终生成的 nginx.conf 配置文件，所以一旦 Watch 到了任何变化控制器都必须根据集群的状态重建一个新的模型，并将其与当前的模型进行比较，如果模型相同则就可以避免生成新的 Nginx 配置并触发重新加载，否则还需要检查模型的差异是否只和端点有关，如果是这样，则然后需要使用 HTTP POST 请求将新的端点列表发送到在 Nginx 内运行的 Lua 处理程序，并再次避免生成新的 Nginx 配置并触发重新加载，如果运行和新模型之间的差异不仅仅是端点，那么就会基于新模型创建一个新的 Nginx 配置了，这样构建模型最大的一个好处就是在状态没有变化时避免不必要的重新加载，可以节省大量 Nginx 重新加载。

下面简单描述了需要重新加载的一些场景：

- 创建了新的 Ingress 资源
- TLS 添加到现有 Ingress
- 从 Ingress 中添加或删除 path 路径
- Ingress、Service、Secret 被删除了
- Ingress 的一些缺失引用对象变可用了，例如 Service 或 Secret
- 更新了一个 Secret

对于集群规模较大的场景下频繁的对 Nginx 进行重新加载显然会造成大量的性能消耗，所以要尽可能减少出现重新加载的场景。

安装

由于 ingress-nginx 所在的节点需要能够访问外网，这样域名可以解析到这些节点上直接使用，所以可以让 ingress-nginx 绑定节点的 80 和 443 端口，所以可以使用 hostPort 或者 hostNetwork 模式，当然对于线上环境来说为了保证高可用，一般是需要运行多个 ingress-nginx 实例，然后可以用一个 nginx/haproxy 作为入口，通过 keepalived 来访问边缘节点的 vip 地址。

> 所谓边缘节点即集群内部用来向集群外暴露服务能力的节点，集群外部的服务通过该节点来调用集群内部的服务，边缘节点是集群内外交流的一个 Endpoint。

安装 ingress-nginx 有多种方式，我们这里直接使用下面的命令进行一键安装：

```
➜ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-
nginx/controller-v1.5.1/deploy/static/provider/cloud/deploy.yaml
# 可以替换对应的两个镜像
# cnych/ingress-nginx:v1.5.1
# cnych/ingress-nginx-kube-webhook-certgen:v20220916-gd32f8c343
```

上面的命令执行后会自动创建一个名为 ingress-nginx 的命名空间，会生成如下几个 Pod：

```
➜ kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-admission-create-2t6gc        0/1     Completed   0          77s
ingress-nginx-admission-patch-dzhzv         0/1     Completed   0          77s
ingress-nginx-controller-68b46f9864-zs8k9   1/1     Running     0          77s
```

此外还会创建如下两个 Service 对象：

```
➜ kubectl get svc -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
                    AGE
ingress-nginx-controller             LoadBalancer   10.96.197.15   <pending>
80:30877/TCP,443:30615/TCP   2m57s
ingress-nginx-controller-admission   ClusterIP      10.96.254.41   <none>        443/TCP
                    2m57s
```

其中 ingress-nginx-controller-admission 是为准入控制器提供服务的，我们也是强烈推荐开启该准入控制器，这样当我们创建不合要求的 Ingress 对象后就会直接被拒绝了，另外一个 ingress-nginx-controller 就是 ingress 控制器对外暴露的服务，我们可以看到默认是一个 LoadBalancer 类型的 Service，我们知道该类型是用于云服务商的，我们这里在本地环境，暂时不能使用，但是可以通过他的 NodePort 来对外暴露，后面我们会提供在本地测试环境提供 LoadBalancer 的方式。

到这里 ingress-nginx 就部署成功了，安装完成后还会创建一个名为 nginx 的 IngressClass 对象：

```
➜ kubectl get ingressclass
NAME    CONTROLLER             PARAMETERS   AGE
nginx   k8s.io/ingress-nginx   <none>       7m14s
➜ kubectl get ingressclass nginx -o yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  labels:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
    app.kubernetes.io/version: 1.5.1
  name: nginx
spec:
  controller: k8s.io/ingress-nginx
```

这里我们只提供了一个 controller 属性，对应的值和 ingress-nginx 的启动参数中的 controller-class 一致的。

```
- args:
    - /nginx-ingress-controller
    - --publish-service=$(POD_NAMESPACE)/ingress-nginx-controller
    - --election-id=ingress-nginx-leader
    - --controller-class=k8s.io/ingress-nginx
    - --ingress-class=nginx
    - --configmap=$(POD_NAMESPACE)/ingress-nginx-controller
    - --validating-webhook=:8443
    - --validating-webhook-certificate=/usr/local/certificates/cert
    - --validating-webhook-key=/usr/local/certificates/key
```

第一个示例

安装成功后，现在我们来为一个 nginx 应用创建一个 Ingress 资源，如下所示：

```
# my-nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      app: my-nginx
  template:
    metadata:
      labels:
        app: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    app: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
      name: http
```

```yaml
  selector:
    app: my-nginx
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-nginx
  namespace: default
spec:
  ingressClassName: nginx # 使用 nginx 的 IngressClass（关联的 ingress-nginx 控制器）
  rules:
    - host: first-ingress.172.18.0.2.nip.io # 将域名映射到 my-nginx 服务
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service: # 将所有请求发送到 my-nginx 服务的 80 端口
                name: my-nginx
                port:
                  number: 80
# 不过需要注意大部分Ingress控制器都不是直接转发到Service
# 而是只是通过Service来获取后端的Endpoints列表，直接转发到Pod，这样可以减少网络跳转，提高性能
```

注意我们这里配置的域名是 first-ingress.172.18.0.2.nip.io，该地址其实会直接映射到 172.18.0.2 上面，该 IP 地址就是我的 Node 节点地址，因为我们这里 ingress 控制器是通过 NodePort 对外进行暴露的，所以可以通过 域名:nodePort 来访问服务。nip.io 是由 PowerDNS 提供支持的开源服务，允许我们可以直接通过使用以下格式将任何 IP 地址映射到主机名，这样我们就不需要在 etc/hosts 文件中配置映射了，对于 Ingress 测试非常方便。

- **10.0.0.1.nip.io** maps to **10.0.0.1**
- **192-168-1-250.nip.io** maps to **192.168.1.250**
- **0a000803.nip.io** maps to **10.0.8.3**

With a name:

- **app.10.8.0.1.nip.io** maps to **10.8.0.1**
- **app-116-203-255-68.nip.io** maps to **116.203.255.68**
- **app-c0a801fc.nip.io** maps to **192.168.1.252**
- **customer1.app.10.0.0.1.nip.io** maps to **10.0.0.1**
- **customer2-app-127-0-0-1.nip.io** maps to **127.0.0.1**
- **customer3-app-7f000101.nip.io** maps to **127.0.1.1**

nip.io maps **<anything>[.-]<IP Address>.nip.io** in "dot", "dash" or "hexadecimal" notation to the corresponding **<IP Address>**:

- dot notation: **magic.127.0.0.1.nip.io**
- dash notation: **magic-127-0-0-1.nip.io**
- hexadecimal notation: **magic-7f000001.nip.io**

这里直接创建上面的资源对象即可：

```
➜ kubectl apply -f my-nginx.yaml
deployment.apps/my-nginx created
service/my-nginx created
ingress.networking.k8s.io/my-nginx created
➜ kubectl get ingress
NAME       CLASS   HOSTS                              ADDRESS   PORTS   AGE
my-nginx   nginx   first-ingress.172.18.0.2.nip.io              80      4m45s
```

在上面的 Ingress 资源对象中我们使用配置 ingressClassName: nginx 指定让我们安装的 ingress-nginx 这个控制器来处理我们的 Ingress 资源，配置的匹配路径类型为前缀的方式去匹配 /，将来自域名 first-ingress.172.18.0.2.nip.io 的所有请求转发到 my-nginx 服务的后端 Endpoints 中去，注意访问的时候需要带上 NodePort 端口。

```
# curl first-ingress.172.18.0.2.nip.io:30877
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

前面我们也提到了 ingress-nginx 控制器的核心原理就是将我们的 Ingress 这些资源对象映射翻译成 Nginx 配置文件 nginx.conf，我们可以通过查看控制器中的配置文件来验证这点：

```
➜ kubectl exec -it ingress-nginx-controller-68b46f9864-zs8k9 -n ingress-nginx -- cat
/etc/nginx/nginx.conf

......
upstream upstream_balancer {
        ### Attention!!!
        #
        # We no longer create "upstream" section for every backend.
        # Backends are handled dynamically using Lua. If you would like to debug
```

```
        # and see what backends ingress-nginx has in its memory you can
        # install our kubectl plugin https://kubernetes.github.io/ingress-nginx/kubectl-
plugin.
        # Once you have the plugin you can use "kubectl ingress-nginx backends" command
to
        # inspect current backends.
        #
        ###

        server 0.0.0.1; # placeholder

        balancer_by_lua_block {
                balancer.balance()
        }

        keepalive 320;
        keepalive_time 1h;
        keepalive_timeout  60s;
        keepalive_requests 10000;

}

......
## start server first-ingress.172.18.0.2.nip.io
server {
        server_name first-ingress.172.18.0.2.nip.io ;

        listen 80  ;
        listen [::]:80  ;
        listen 443  ssl http2 ;
        listen [::]:443  ssl http2 ;

        set $proxy_upstream_name "-";

        ssl_certificate_by_lua_block {
                certificate.call()
        }

        location / {

                set $namespace      "default";
                set $ingress_name   "my-nginx";
                set $service_name   "my-nginx";
                set $service_port   "80";
                set $location_path  "/";
                set $global_rate_limit_exceeding n;

                rewrite_by_lua_block {
                        lua_ingress.rewrite({
                                force_ssl_redirect = false,
```

```
                                ssl_redirect = true,
                                force_no_ssl_redirect = false,
                                preserve_trailing_slash = false,
                                use_port_in_redirects = false,
                                global_throttle = { namespace = "", limit = 0,
window_size = 0, key = { }, ignored_cidrs = { } },
                        })
                        balancer.rewrite()
                        plugins.run()
                }

                # be careful with `access_by_lua_block` and `satisfy any` directives as
satisfy any
                # will always succeed when there's `access_by_lua_block` that does not
have any lua code doing `ngx.exit(ngx.DECLINED)`
                # other authentication method such as basic auth or external auth
useless - all requests will be allowed.
                #access_by_lua_block {
                #}

                header_filter_by_lua_block {
                        lua_ingress.header()
                        plugins.run()
                }

                body_filter_by_lua_block {
                        plugins.run()
                }

                log_by_lua_block {
                        balancer.log()

                        monitor.call()

                        plugins.run()
                }

                port_in_redirect off;

                set $balancer_ewma_score -1;
                set $proxy_upstream_name "default-my-nginx-80";
                set $proxy_host          $proxy_upstream_name;
                set $pass_access_scheme  $scheme;

                set $pass_server_port    $server_port;

                set $best_http_host      $http_host;
                set $pass_port           $pass_server_port;

                set $proxy_alternative_upstream_name "";
```

```
                client_max_body_size                    1m;

                proxy_set_header Host                   $best_http_host;

                # Pass the extracted client certificate to the backend

                # Allow websocket connections
                proxy_set_header                        Upgrade           $http_upgrade;

                proxy_set_header                        Connection
 $connection_upgrade;

                proxy_set_header X-Request-ID           $req_id;
                proxy_set_header X-Real-IP              $remote_addr;

                proxy_set_header X-Forwarded-For        $remote_addr;

                proxy_set_header X-Forwarded-Host       $best_http_host;
                proxy_set_header X-Forwarded-Port       $pass_port;
                proxy_set_header X-Forwarded-Proto      $pass_access_scheme;
                proxy_set_header X-Forwarded-Scheme     $pass_access_scheme;

                proxy_set_header X-Scheme               $pass_access_scheme;

                # Pass the original X-Forwarded-For
                proxy_set_header X-Original-Forwarded-For $http_x_forwarded_for;

                # mitigate HTTPoxy Vulnerability
                # https://www.nginx.com/blog/mitigating-the-httpoxy-vulnerability-with-
nginx/
                proxy_set_header Proxy                  "";

                # Custom headers to proxied server

                proxy_connect_timeout                   5s;
                proxy_send_timeout                      60s;
                proxy_read_timeout                      60s;

                proxy_buffering                         off;
                proxy_buffer_size                       4k;
                proxy_buffers                           4 4k;

                proxy_max_temp_file_size                1024m;

                proxy_request_buffering                 on;
                proxy_http_version                      1.1;

                proxy_cookie_domain                     off;
                proxy_cookie_path                       off;
```

```
                # In case of errors try the next upstream server before returning an
error
                proxy_next_upstream                     error timeout;
                proxy_next_upstream_timeout             0;
                proxy_next_upstream_tries               3;

                proxy_pass http://upstream_balancer;

                proxy_redirect                          off;

        }

}
## end server first-ingress.172.18.0.2.nip.io
......
```

我们可以在 nginx.conf 配置文件中看到上面我们新增的 Ingress 资源对象的相关配置信息，不过需要注意的是现在并不会为每个 backend 后端都创建一个 upstream 配置块，现在是使用 Lua 程序进行动态处理的，所以我们没有直接看到后端的 Endpoints 相关配置数据。

此外我们也可以安装一个 kubectl 插件 https://kubernetes.github.io/ingress-nginx/kubectl-plugin 来辅助使用 ingress-nginx，要安装该插件的前提需要先安装 krew，然后执行下面的命令即可：

```
➜ kubectl krew install ingress-nginx
```