

k8s学习笔记（7）---kubernetes核心组件之apiserver详解

kubernetes核心组件之apiserver详解

1、API Server简介

k8s API Server提供了k8s各类资源对象（pod,RC,Service等）的增删改查及watch等HTTP Rest接口，是整个系统的数据总线和数据中心。

1.1 API Server的功能

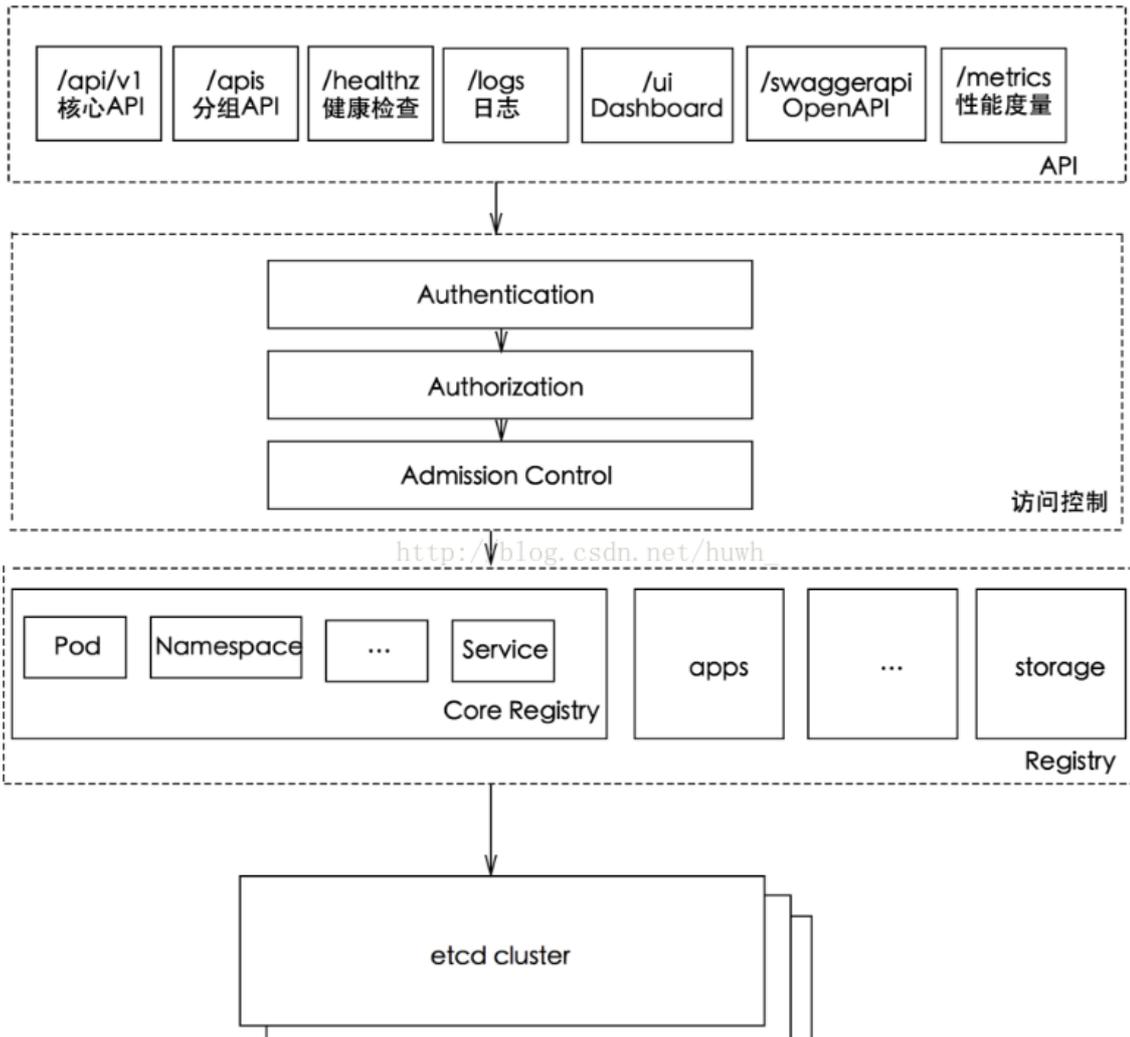
- 提供了集群管理的REST API接口(包括认证授权、数据校验以及集群状态变更);
- 提供其他模块之间的数据交互和通信的枢纽（其他模块通过API Server查询或修改数据，只有API Server才直接操作etcd）；
- 是资源配额控制的入口；
- 拥有完备的集群安全机制;

1.2 kube-apiserver工作原理

kube-apiserver提供了k8s的rest api，实现了认证、授权和准入控制等安全功能，同时也负责了集群状态的存储操作。

- rest api
 - kube-apiserver支持同时提供https和http api，其中http api是非安全接口，不做任何认证授权机制，不建议生产环境启用，但两个接口提供的rest api格式相同。
 - https api 默认监听在6443端口（- secure-port=6443）；
 - http api 默认监听在127.0.0.1的8080端口（使用参数 --insecure-port=8080）；
- 访问控制说明
 - k8s api 每个请求都会经过多阶段的访问控制才会被接受，包括认证、授权及准入控制等。
 - 认证
 - 开启TLS情况下，所有请求都需要首先认证。k8s支持多种认证机制，并且支持同时开启多个认证插件（仅一个认证通过即可），如认证成功，则用户的username会传入授权模块做进一步的授权验证，而认证失败的请求则返回http 401。
 - 授权
 - 认证之后的请求就到了授权模块，和认证类似，k8s也支持多种授权机制，并支持同时开启多个授权插件（仅一个验证通过即可）。如授权成功，则用户的请求会发送到准入控制模块做进一步的请求验证，失败的请求则返回http403。
 - 准入控制
 - 用来对请求做进一步的验证或添加默认参数，不同于认证和授权只关心请求的用户和操作，准入控制还会处理请求的内容，并且仅对创建、更新、删除或连接（如代理）等有效，而对读操作无效。准入控制也支持同时开启多个插件，但他们是依次调用的，只有全部插件都通过的请求才可以允许进入系统。

kube-apiserver工作原理图如下：



1.3 访问kubernetes API

k8s通过kube-apiserver这个进程提供服务，该进程运行在单个k8s-master节点上，默认有两个端口。

- 本地端口
 - 1.该端口用于接收HTTP请求；
 - 2.该端口默认值为8080，可以通过API Server的启动参数“- insecure-port”的值来修改默认值；
 - 3.默认的IP地址为“localhost”，可以通过启动参数“- insecure-bind-address”的值来修改该IP地址；
 - 4.非认证或授权的HTTP请求通过该端口访问API Server；
- 安全端口
 - 1.该端口默认值为6443，可通过启动参数“- secure-port”的值来修改默认值；
 - 2.默认IP地址为非本地（Non-Localhost）网络端口，通过启动参数“- 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.默认不启动HTTPS安全访问控制

1 直接访问api

通过本地8080端口使用curl命令能直接获取REST api的信息,例如：

查看版本:

```
curl 127.0.0.1:8080/api
```

查看支持的资源对象:

```
curl 127.0.0.1:8080/api/v1
```

查看资源对象信息:

```
curl 127.0.0.1:8080/api/v1/nodes
```

```
curl 127.0.0.1:8080/api/v1/pods
```

```
curl 127.0.0.1:8080/api/v1/services
```

```
curl 127.0.0.1:8080/api/v1/replicationcontrollers
```

具体到某一个具体的对象:

```
curl 127.0.0.1:8080/api/v1/nodes/10.0.0.3
```

指定不同namespace中的对象进行访问:

```
curl http://127.0.0.1:8080/api/v1/namespaces/default/services/php-apache
```

直接访问后端服务内容:

```
curl http://127.0.0.1:8080/api/v1/namespaces/default/services/http:tomcat-service:/proxy/
```

```
curl http://127.0.0.1:8080/api/v1/namespaces/default/services/http:php-apache:/proxy/
```

其他的API信息:

```
curl 127.0.0.1:8080/apis/batch/v1
```

2 Kubernetes Proxy

Kubernetes Proxy主要用于代理REST请求。可以通过这种代理方式将API Server收到的REST请求转发到某个Node上的kubectl上,由Kubelet负责响应。

创建一个Proxy 接口(此处创建一个监听本地10.0.0.1上的8001 端口):

```
kubectl proxy --port=8001 --accept-hosts='.*' --address='10.0.0.1' &
```

如果需要对访问的资源 and 源地址进行限制,可以使用正则进行匹配,限制对services进行访问:

```
kubectl proxy --port=8001 --accept-hosts='.*' --address=10.0.0.1 --reject-paths='^/api/v1/services'
```

通过此端口可以在外部直接访问此api代理,在新的版本中,访问方式和常规访问的URL一致:

```
curl http://10.0.0.1:8001/api/v1/pods
```

```
curl http://10.0.0.1:8001/api/v1/nodes
```

```
curl http://10.0.0.1:8001/api/v1/services
```

```
curl http://10.0.0.1:8001/api/v1/replicationcontrollers
```

```
curl http://10.0.0.1:8001/api/v1/namespaces/default/pods/tomcat-deployment-65799d5fc4-5dx4h
```

```
curl http://10.0.0.1:8001/api/v1/namespaces/default/services/http:php-apache:/proxy/
```

```
curl 10.0.0.1:8001/apis/batch/v1
```

3 kubectl客户端

命令行工具kubectl客户端,通过命令行参数转换为对API Server的REST API调用,并将调用结果输出。

命令格式: `kubectl [command] [options]`

4 编程方式调用

使用场景:

- 运行在Pod里的用户进程调用kubernetes API,通常用来实现分布式集群搭建的目标;
- 开发基于kubernetes的管理平台,比如调用kubernetes API来完成Pod、Service、RC等资源对象的图形化创建和管理界面。可以使用kubernetes提供的Client Library;
具体可参考

1.4 API server和集群模块的交互

API Server作为集群的核心,负责集群各个功能模块之间的通信,集群中的各个功能模块通过API Server将信息存入etcd数据库中,获取这些数据就通过API Server提供的REST接口通过使用GET,LIST,或者Watch方法,从而实现对各个模块之间的交互。

为了缓解各模块对API Server的访问压力，各功能模块都采用缓存机制来缓存数据，各功能模块定时从API Server获取指定的资源对象信息（LIST/WATCH方法），然后将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。

1.4.1 API Server与kubelet的交互

每个Node节点上的kubelet定期就会调用API Server的REST接口报告自身状态，API Server接收这些信息后，将节点状态信息更新到etcd中。kubelet也通过API Server的Watch接口监听Pod信息，从而对Node机器上的POD进行管理。

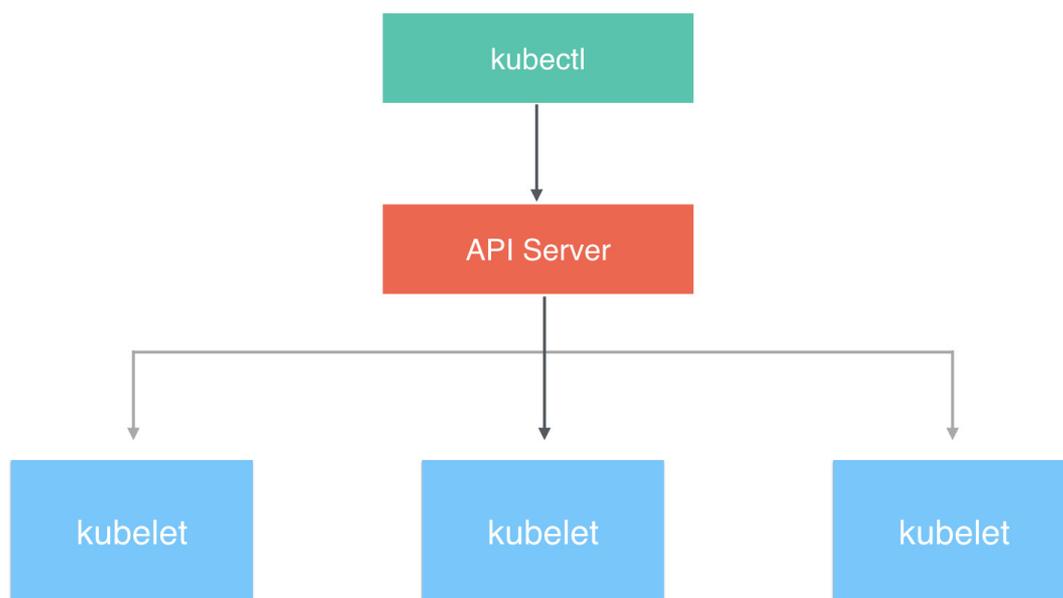
监听信息	kubelet动作
新的POD副本被调度绑定到本节点	执行POD对应的容器的创建和启动逻辑
POD对象被删除	删除本节点上相应的POD容器
修改POD信息	修改本节点的POD容器

1.4.2 API Server与kube-controller-manager交互

kube-controller-manager中的Node Controller模块通过API Server提供的Watch接口，实时监控Node的信息，并做相应处理。

1.4.3 API Server与kube-scheduler交互

Scheduler通过API Server的Watch接口监听到新建Pod副本的信息后，它会检索所有符合该Pod要求的Node列表，开始执行Pod调度逻辑。调度成功后将Pod绑定到目标节点上。



2、API Server启动参数详解

API Server 主要是和 etcd 打交道，并且对外提供 HTTP 服务，以及进行安全控制，因此它的命令行提供的参数也主要和这几个方面有关。下面是一些比较重要的参数以及说明（不同版本参数可能会有不同）：

参数	描述
<code>--advertise-address</code>	向集群成员发布apiserver的IP地址，该地址必须能够被集群的成员访问。如果为空，则使用 <code>--bind-address</code> ，如果 <code>--bind-address</code> 未指定，那么使用主机的默认接口。
<code>--authorization-mode</code>	授权模式，安全接口上的授权。默认值：AlwaysAllow

参数	描述
<code>--allow-privileged</code>	以逗号分隔的列表: AlwaysAllow,AlwaysDeny,ABAC,Webhook,RBAC,Node 监听安全端口的IP地址。必须能被集群的其他以及CLI/web客户机访问
<code>--tls-cert-file</code>	包含HTTPS的默认x509证书的文件。CA证书, 如果有的话, 在服务器证书之后连接。如果启用了HTTPS服务, 但未提供 <code>--tls-cert-file</code> 和 <code>-tls-private-key-file</code> , 则会为公共地址生成自签名证书和密钥, 并将其保存到 <code>-cert-dir</code> 指定的目录中。
<code>--cert-dir</code>	TLS 证书的存放目录,默认为/var/run/kubernetes
<code>--token-auth-file</code>	使用该文件在安全端口通过token身份验证来保护API服务
<code>--admission-control</code>	准入控制过程分两个阶段。第一阶段, 运行 mutating admission controllers。第二阶段, 运行 validating admission controllers。有些控制器会运行两次。如果任一阶段中的任何控制器拒绝请求, 则立即拒绝整个请求, 并向最终用户返回错误。默认值为 AlwaysAdmit 。 AlwaysAdmit, AlwaysDeny, AlwaysPullImages, DefaultStorageClass, DefaultTolerationSeconds, DenyEscalatingExec, DenyExecOnPrivileged, EventRateLimit, ExtendedResourceToleration, ImagePolicyWebhook, Initializers, LimitPodHardAntiAffinityTopology, LimitRanger, MutatingAdmissionWebhook, NamespaceAutoProvision, NamespaceExists, NamespaceLifecycle, NodeRestriction, OwnerReferencesPermissionEnforcement, PersistentVolumeClaimResize, PersistentVolumeLabel, PodNodeSelector, PodPreset, PodSecurityPolicy, PodTolerationRestriction, Priority, ResourceQuota, SecurityContextDeny, ServiceAccount, StorageObjectInUseProtection, TaintNodesByCondition, ValidatingAdmissionWebhook
<code>--advertise-address</code>	向集群成员发布apiserver的IP地址, 该地址必须能够被集群的成员访问。如果为空, 则使用 <code>--bind-address</code> , 如果 <code>--bind-address</code> 未指定, 那么使用主机的默认接口。
<code>--kubelet-https=true</code>	kubelet通信使用https, 默认值 true
<code>--service-cluster-ip-range</code>	CIDR表示IP范围, 用于分配服务集群IP。不能与分配给pod节点的IP重叠 (default 10.0.0.0/24)
<code>--allow-privileged</code>	true允许特权模式的容器。默认值false
<code>--insecure-bind-address</code>	HTTP 访问的地址(default 127.0.0.1)
<code>--insecure-port</code>	HTTP 访问的端口(default 8080)
<code>--bind-address</code>	HTTPS 安全接口的监听地址, 必须能被集群的其他以及CLI/web客户机访问, 默认为0.0.0.0
<code>--secure-port</code>	HTTPS 安全接口的监听端口,默认为6443
<code>--etcd-prefix</code>	要在etcd中所有资源路径之前添加的前缀,默认为 “/registry”
<code>--etcd-servers</code>	etcd服务器列表 (格式: //ip:port), 逗号分隔
<code>--log-dir</code>	日志存放的目录
<code>--alsologtostderr</code>	日志信息同时输出到stderr及文件
<code>--logtostderr</code>	日志信息输出到stderr 而不是文件

3、API Server安装和运行

安装（可直接下载二进制文件或kube-apiserver镜像）

二进制文件下载如下：

```
$ wget https://storage.googleapis.com/kubernetes-release/release/v1.0.3/bin/linux/amd64/kube-apiserver
$ chmod +x kube-apiserver
```

运行

API Server 是通过提供的 kube-apiserver 二进制文件直接运行的，下面的例子指定了 service 分配的 ip 范围，etcd 的地址，和对外提供服务的 ip 地址：

```
/usr/bin/kube-apiserver \
--service-cluster-ip-range=10.20.0.1/24 \
--etcd-servers=http://127.0.0.1:2379 \
--advertise-address=192.168.8.100 \
--bind-address=192.168.8.100 \
--insecure-bind-address=192.168.8.100 \
--v=4
```

直接访问 8080 端口，API Server 会返回它提供了哪些接口：

```
curl http://192.168.8.100:8080
```

```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/apps",
    "/apis/apps/v1alpha1",
    "/apis/autoscaling",
    "/apis/autoscaling/v1",
    "/apis/batch",
    "/apis/batch/v1",
    "/apis/batch/v2alpha1",
    "/apis/extensions",
    "/apis/extensions/v1beta1",
    "/apis/policy",
    "/apis/policy/v1alpha1",
    "/apis/rbac.authorization.k8s.io",
    "/apis/rbac.authorization.k8s.io/v1alpha1",
    "/healthz",
    "/healthz/ping",
    "/logs",
    "/metrics",
    "/swaggerapi",
    "/ui",
    "/version"
  ]
}
```

而目前最重要的路径是 /api/v1，里面包含了 kubernetes 所有资源的操作，比如下面的 nodes：

```
http http://192.168.8.100:8080/api/v1/nodes
```

```
HTTP/1.1 200 OK
```

```
Content-Length: 112
```

```
Content-Type: application/json
```

```
Date: Thu, 08 Sep 2016 08:14:45 GMT
```

```
{  
  
  "apiVersion": "v1",  
  
  "items": [],  
  
  "kind": "NodeList",  
  
  "metadata": {  
  
    "resourceVersion": "12",  
  
    "selfLink": "/api/v1/nodes"  
  }  
}
```

API 以 json 的形式返回，会通过 apiVersion 来说明 API 版本号，kind 说明请求的是什么资源。不过这里面的内容是空的，因为目前还没有任何 kubelet 节点接入到我们的 API Server。对应的，pod 也是空的：

```
http http://192.168.8.100:8080/api/v1/pods
```

```
HTTP/1.1 200 OK
```

```
Content-Length: 110
```

```
Content-Type: application/json
```

```
Date: Thu, 08 Sep 2016 08:18:53 GMT
```

```
{  
  
  "apiVersion": "v1",  
  
  "items": [],  
  
  "kind": "PodList",  
  
  "metadata": {  
  
    "resourceVersion": "12",  
  
    "selfLink": "/api/v1/pods"  
  }  
}
```

添加节点也非常简单，启动 kubelet 的时候使用 --api-servers 指定要接入的 API Server 就行。kubelet 启动之后，会把自己注册到指定的 API Server，然后监听 API 对应 pod 的变化，根据 API 中 pod 的实际信息来管理节点上 pod 的生命周期。

我们可以看到，kubelet 收集了很多关于自身节点的信息，这些信息也会不断更新。这些信息里面不仅包含节点的系统信息（系统架构，操作系统版本，内核版本等）、还有镜像信息（节点上有哪些已经下载的 docker 镜像）、资源信息（Memory 和 Disk 的总量和可用量）、以及状态信息（是否正常，可以分配 pod 等）。

现在访问 `/api/v1/nodes` 就能看到已经添加进来的节点:

```
http http://192.168.8.100:8080/api/v1/nodes
```

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
Date: Thu, 08 Sep 2016 08:27:44 GMT
```

```
Transfer-Encoding: chunked
```

```
{
  "apiVersion": "v1",
  "items": [
    {
      "metadata": {
        "annotations": {
          "volumes.kubernetes.io/controller-managed-attach-detach": "true"
        },
        "creationTimestamp": "2016-09-08T08:23:01Z",
        "labels": {
          "beta.kubernetes.io/arch": "amd64",
          "beta.kubernetes.io/os": "linux",
          "kubernetes.io/hostname": "192.168.8.100"
        },
        "name": "192.168.8.100",
        "resourceVersion": "65",
        "selfLink": "/api/v1/nodes/192.168.8.100",
        "uid": "74e16eba-759d-11e6-b463-080027c09e5b"
      },
      "spec": {
        "externalID": "192.168.8.100"
      },
      "status": {
        "addresses": [
          {
            "address": "192.168.8.100",
            "type": "LegacyHostIP"
          },
          {
            "address": "192.168.8.100",
            "type": "InternalIP"
          }
        ],
        "allocatable": {
          "alpha.kubernetes.io/nvidia-gpu": "0",
          "cpu": "1",
          "memory": "502164Ki",
          "pods": "110"
        },
        "capacity": {
          "alpha.kubernetes.io/nvidia-gpu": "0",
          "cpu": "1",
          "memory": "502164Ki",
          "pods": "110"
        },
        "conditions": [
          {
            "lastHeartbeatTime": "2016-09-08T08:27:36Z",
            "lastTransitionTime": "2016-09-08T08:23:01Z",
            "message": "kubelet has sufficient disk space available",
            "reason": "KubeletHasSufficientDisk",
            "status": "False",
            "type": "OutOfDisk"
          },
          {
            "lastHeartbeatTime": "2016-09-08T08:27:36Z",
            "lastTransitionTime": "2016-09-08T08:23:01Z",

```

```
"message": "kubelet has sufficient memory available",
"reason": "KubeletHasSufficientMemory",
"status": "False",
"type": "MemoryPressure"
},
{
"lastHeartbeatTime": "2016-09-08T08:27:36Z",
"lastTransitionTime": "2016-09-08T08:24:56Z",
"message": "kubelet is posting ready status",
"reason": "KubeletReady",
"status": "True",
"type": "Ready"
}
],
"daemonEndpoints": {
"kubeletEndpoint": {
"Port": 10250
}
},
"images": [
{
"names": [
"172.16.1.41:5000/nginx:latest"
],
"sizeBytes": 425626718
},
{
"names": [
"172.16.1.41:5000/hyperkube:v0.18.2"
],
"sizeBytes": 207121551
},
{
"names": [
"172.16.1.41:5000/etcd:v3.0.4"
],
"sizeBytes": 43302056
},
{
"names": [
"172.16.1.41:5000/busybox:latest"
],
"sizeBytes": 1092588
},
{
"names": [
"172.16.1.41:5000/google_containers/pause:0.8.0"
],
"sizeBytes": 241656
}
],
"nodeInfo": {
"architecture": "amd64",
"bootID": "48955926-11dd-4ad3-8bb0-2585b1c9215d",
"containerRuntimeVersion": "docker://1.10.3",
"kernelVersion": "3.10.0-123.13.1.el7.x86_64",
"kubeProxyVersion": "v1.3.1-beta.0.6+fbf3f3e5292fb0",
"kubeletVersion": "v1.3.1-beta.0.6+fbf3f3e5292fb0",
"machineID": "b9597c4ae5f24494833d35e806e00b29",
"operatingSystem": "linux",
"osImage": "CentOS Linux 7 (Core)",
"systemUUID": "823EB67A-057E-4EFF-AE7F-A758140CD2F7"
}
}
}
```

```
],
"kind": "NodeList",
"metadata": {
"resourceVersion": "65",
"selfLink": "/api/v1/nodes"
}
}
```

使用 curl 执行 POST 请求，设置头部内容为 application/json，传过去文件中的 json 值，可以看到应答(其中 status 为 pending，表示以及接收到请求，正在准备处理)：

```
curl -s -X POST -H "Content-Type: application/json" http://192.168.8.100:8080/api/v1/namespaces/default/pods --data @nginx_pod.json
```

```
{
"kind": "Pod",
"apiVersion": "v1",
"metadata": {
"name": "nginx-server",
"namespace": "default",
"selfLink": "/api/v1/namespaces/default/pods/nginx-server",
"uid": "888e95d0-75a9-11e6-b463-080027c09e5b",
"resourceVersion": "573",
"creationTimestamp": "2016-09-08T09:49:28Z"
},
"spec": {
"volumes": [
{
"name": "nginx-logs",
"emptyDir": {}
}
],
"containers": [
{
"name": "nginx-server",
"image": "172.16.1.41:5000/nginx",
"ports": [
{
"containerPort": 80,
"protocol": "TCP"
}
],
"resources": {},
"volumeMounts": [
{
"name": "nginx-logs",
"mountPath": "/var/log/nginx"
}
],
"terminationMessagePath": "/dev/termination-log",
"imagePullPolicy": "Always"
}
],
"restartPolicy": "Always",
"terminationGracePeriodSeconds": 30,
"dnsPolicy": "ClusterFirst",
"nodeName": "192.168.8.100",
"securityContext": {}
},
"status": {
"phase": "Pending"
}
}
```

返回中包含了我们提交 pod 的信息，并且添加了 status、metadata 等额外信息。

一段时间去查询 pod, 就可以看到 pod 的状态已经更新了:

```
http http://192.168.8.100:8080/api/v1/namespaces/default/pods
```

```
HTTP/1.1 200 OK
```

```
Content-Type: application/json
```

```
Date: Thu, 08 Sep 2016 09:51:29 GMT
```

```
Transfer-Encoding: chunked
```

```
{
  "apiVersion": "v1",
  "items": [
    {
      "metadata": {
        "creationTimestamp": "2016-09-08T09:49:28Z",
        "name": "nginx-server",
        "namespace": "default",
        "resourceVersion": "592",
        "selfLink": "/api/v1/namespaces/default/pods/nginx-server",
        "uid": "888e95d0-75a9-11e6-b463-080027c09e5b"
      },
      "spec": {
        "containers": [
          {
            "image": "172.16.1.41:5000/nginx",
            "imagePullPolicy": "Always",
            "name": "nginx-server",
            "ports": [
              {
                "containerPort": 80,
                "protocol": "TCP"
              }
            ],
            "resources": {},
            "terminationMessagePath": "/dev/termination-log",
            "volumeMounts": [
              {
                "mountPath": "/var/log/nginx",
                "name": "nginx-logs"
              }
            ]
          },
          {
            "args": [
              "-c",
              "tail -f /logdir/access.log"
            ],
            "command": [
              "bin/sh"
            ],
            "image": "172.16.1.41:5000/busybox",
            "imagePullPolicy": "Always",
            "name": "log-output",
            "resources": {},
            "terminationMessagePath": "/dev/termination-log",
            "volumeMounts": [
              {
                "mountPath": "/logdir",
                "name": "nginx-logs"
              }
            ]
          }
        ],
        "dnsPolicy": "ClusterFirst",
        "nodeName": "192.168.8.100",
        "restartPolicy": "Always",

```

```
"securityContext": {},
"terminationGracePeriodSeconds": 30,
"volumes": [
{
"emptyDir": {},
"name": "nginx-logs"
}
],
"status": {
"conditions": [
{
"lastProbeTime": null,
"lastTransitionTime": "2016-09-08T09:49:28Z",
"status": "True",
"type": "Initialized"
},
{
"lastProbeTime": null,
"lastTransitionTime": "2016-09-08T09:49:44Z",
"status": "True",
"type": "Ready"
},
{
"lastProbeTime": null,
"lastTransitionTime": "2016-09-08T09:49:44Z",
"status": "True",
"type": "PodScheduled"
}
],
"containerStatuses": [
{
"containerID": "docker://8b79eeea60f27b6d3f0a19cbd1b3ee3f83709bcf56574a6e1124c69a6376972d",
"image": "172.16.1.41:5000/busybox",
"imageID": "docker://sha256:8c566faa3abdaebc33d40c1b5e566374c975d17754c69370f78c00c162c1e075",
"lastState": {},
"name": "log-output",
"ready": true,
"restartCount": 0,
"state": {
"running": {
"startedAt": "2016-09-08T09:49:43Z"
}
}
},
{
"containerID": "docker://96e64cdba7b05d4e30710a20e958ff5b8f1f359c8d16d32622b36f0df0cb353c",
"image": "172.16.1.41:5000/nginx",
"imageID": "docker://sha256:51d764c1fd358ce81fd0e728436bd0175ff1f3fd85fc5d1a2f9ba3e7dc6bbaf6",
"lastState": {},
"name": "nginx-server",
"ready": true,
"restartCount": 0,
"state": {
"running": {
"startedAt": "2016-09-08T09:49:36Z"
}
}
}
],
"hostIP": "192.168.8.100",
"phase": "Running",
"podIP": "172.17.0.2",
"startTime": "2016-09-08T09:49:28Z"
}
```

```
}  
],  
"kind": "PodList",  
  
"metadata": {  
"resourceVersion": "602",  
"selfLink": "/api/v1/namespaces/default/pods"  
}  
}
```

可以看到 pod 已经在运行，并且给分配了 ip: 172.17.0.2，通过 curl 也可以访问它的服务：

```
curl -s http://172.17.0.2 | head -n 5  
  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx on Debian!</title>  
<style>
```