kubernetes 结构分析

参考资源

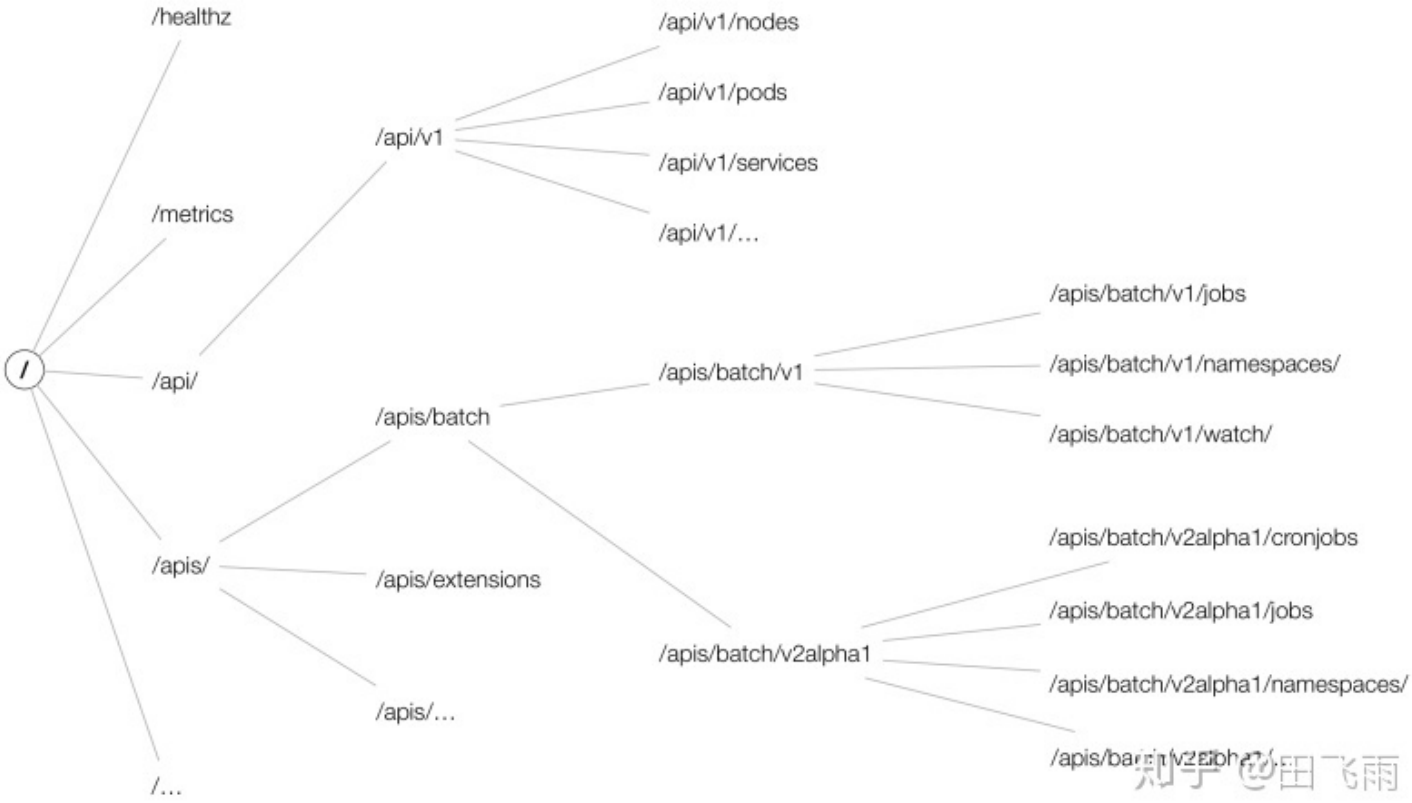
kube-apiserver

kube-apiserver 是 kubernetes 中与 etcd 直接交互的一个组件，其控制着 kubernetes 中核心资源的变化。它主要提供了以下几个功能：

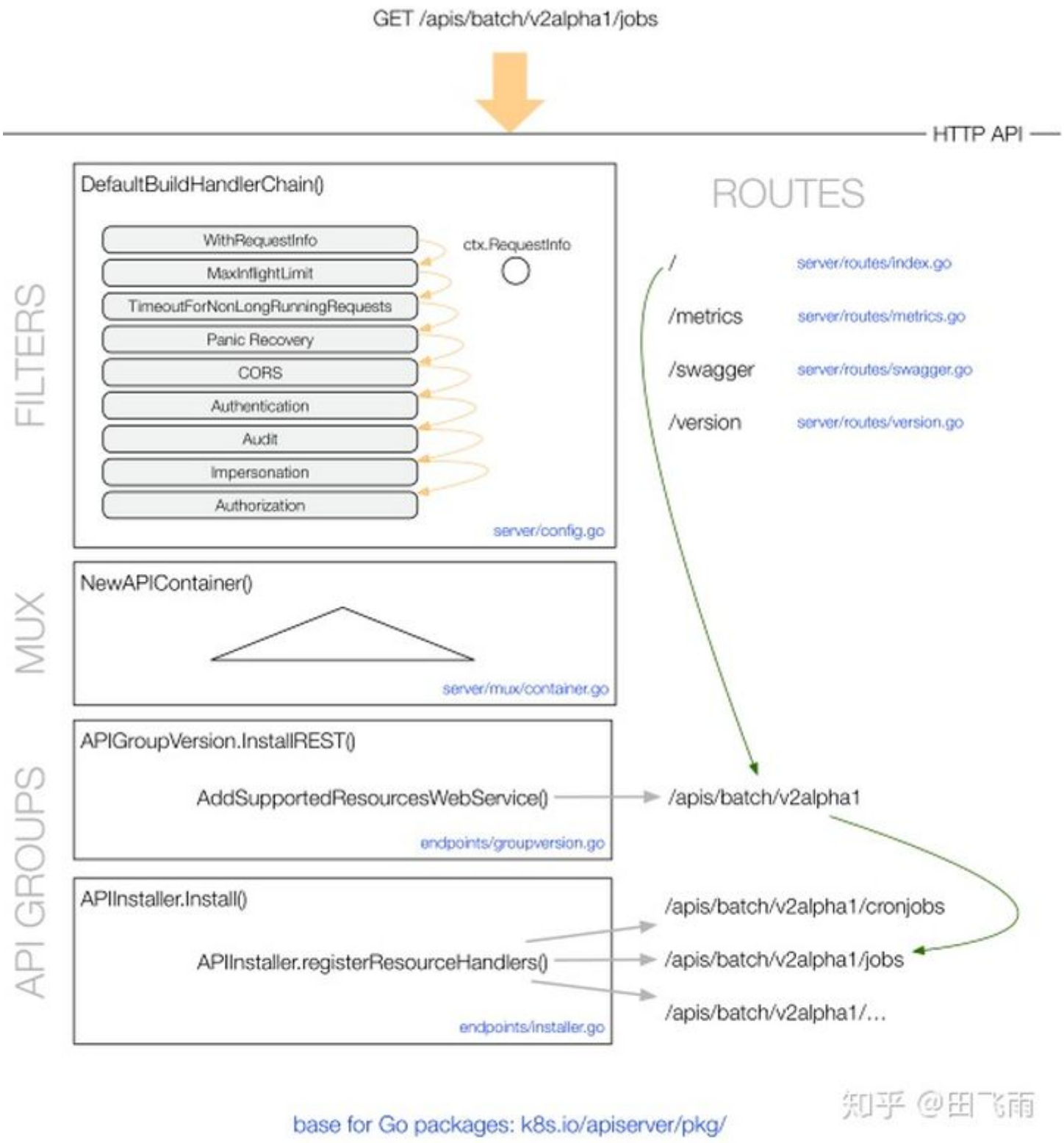
- 提供 Kubernetes API，包括认证授权、数据校验以及集群状态变更等，供客户端及其他组件调用；
- 代理集群中的一些附加组件组件，如 Kubernetes UI、metrics-server、npd 等；
- 创建 kubernetes 服务，即提供 apiserver 的 Service，kubernetes Service；
- 资源在不同版本之间的转换；

kube-apiserver 主要通过对外提供 API 的方式与其他组件进行交互，可以调用 kube-apiserver 的接口 $ curl -k https://:6443或者通过其提供的 swagger-ui 获取到，其主要有以下三种 API：

- core group：主要在 /api/v1 下；
- named groups：其 path 为 /apis/$NAME/$VERSION；
- 暴露系统状态的一些 API：如/metrics 、/healthz 等；

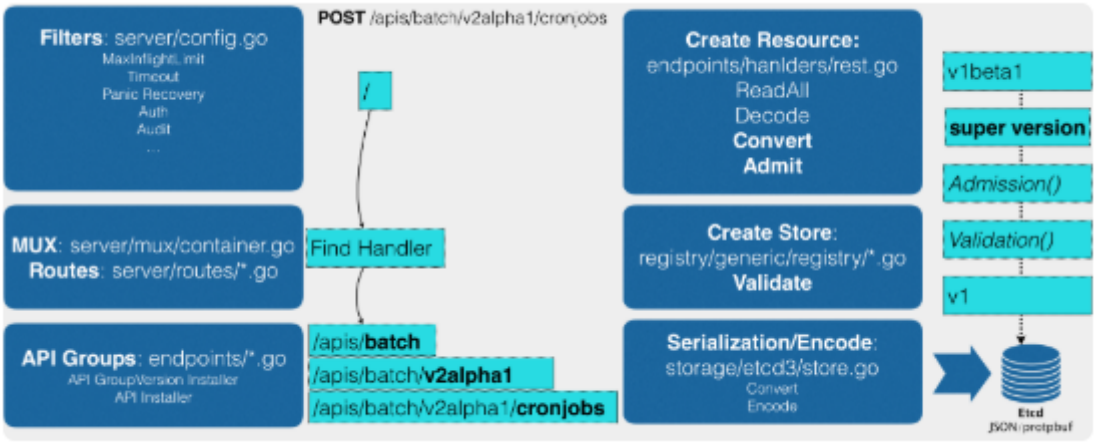


此处以一次 POST 请求示例说明，当请求到达 kube-apiserver 时，kube-apiserver 首先会执行在 http filter chain 中注册的过滤器链，该过滤器对其执行一系列过滤操作，主要有认证、鉴权等检查操作。当 filter chain 处理完成后，请求会通过 route 进入到对应的 handler 中，handler 中的操作主要是与 etcd 的交互，在 handler 中的主要的操作如下所示



- APIServer首先过滤这个请求并完成一些前置性工作. 如:授权,超时处理,审计
- 然后请求进入MUX和Routers流程.这个步骤是APIServer完成URL和Handler绑定的场所.APIServer的Handler的功能就是找到用户提交资源对应的类型定义
- API根据对应的类型定义配合用户提交的YAML文件里的字段.在内存中创建出一个相应的对象.在创建的过程中APIServer会进行一个Convert操作,把用户提交的YAML文件转换成一个叫Super Version的对象.它是该API资源类型所有版本字段的全集.这样用户提交不同的版本的YAML文件就都可以使用Super Version对象来处理

- APIServer会对内存中的对象进行Admission()操作,如Admission Controller和Initializer步骤都属于Admission阶段
- APIServer会把对象进行Validation操作,它负责验证这个对象里的各个字段是否合法.验证通过后的API对象都会保存到APIServer中一个叫 Registry的数据结构中
- APIServer会把验证过的Super API对象转换用户最初提交的版本进行序列化操作并保存到Etcd中.至此整个API对象创建完毕



kube-apiserver 中的组件

kube-apiserver 共由 3 个组件构成（Aggregator、KubeAPIServer、APIExtensionServer），这些组件依次通过 Delegation 处理请求：

- Aggregator：暴露的功能类似于一个七层负载均衡，将来自用户的请求拦截转发给其他服务器，并且负责整个 APIServer 的 Discovery 功能

  > Aggregator 通过 APIServices 对象关联到某个 Service 来进行请求的转发，其关联的 Service 类型进一步决定了请求转发形式。Aggregator 包括一个 GenericAPIServer 和维护自身状态的 Controller。其中 GenericAPIServer 主要处理 apiregistration.k8s.io 组下的 APIService 资源请求。

- KubeAPIServer ： 负责对请求的一些通用处理，认证、鉴权等，以及处理各个内建资源的 REST 服务；

  > KubeAPIServer 主要是提供对 API Resource 的操作请求，为 kubernetes 中众多 API 注册路由信息，暴露 RESTful API 并且对外提供 kubernetes service，使集群中以及集群外的服务都可以通过 RESTful API 操作 kubernetes 中的资源。

- APIExtensionServer：主要处理 CustomResourceDefinition（CRD）和 CustomResource（CR）的 REST 请求，也是 Delegation 的最后一环，如果对应 CR 不能被处理的话则会返回 404。Aggregator 和 APIExtensionsServer 对应两种主要扩展 APIServer 资源的方式，即分别是 AA 和 CRD。

> APIExtensionServer 作为 Delegation 链的最后一层，是处理所有用户通过 Custom Resource Definition 定义的资源服务器。

kube-controller-manager

NodeLifecycleController

NodeLifecycleController 主要功能是定期监控 node 的状态并根据 node 的 condition 添加对应的 taint 标签或者直接驱逐 node 上的 pod。

job 的基本功能

job 在 kubernetes 中主要用来处理离线任务，job 直接管理 pod，可以创建一个或多个 pod 并会确保指定数量的 pod 运行完成。kubernetes 中有两种类型的 job，分别为 cronjob 和 batchjob，cronjob 类似于定时任务是定时触发的而 batchjob 创建后会直接运行，本文主要介绍 batchjob，下面简称为 job。

GarbageCollectorController

在 kubernetes 中对象的回收操作是由 GarbageCollectorController 负责的，其作用就是当删除一个对象时，会根据指定的删除策略回收该对象及其依赖对象,
kubernetes 中有三种删除策略：Orphan、Foreground 和 Background，三种删除策略的意义分别为：

- Orphan 策略：非级联删除，删除对象时，不会自动删除它的依赖或者是子对象，这些依赖被称作是原对象的孤儿对象，例如当执行以下命令时会使用 Orphan 策略进行删除，此时 ds 的依赖对象 controllerrevision 不会被删除；
  $ kubectl delete ds/nginx-ds --cascade=false
- Background 策略：在该模式下，kubernetes 会立即删除该对象，然后垃圾收集器会在后台删除这些该对象的依赖对象；
- Foreground 策略：在该模式下，对象首先进入“删除中”状态，即会设置对象的 deletionTimestamp 字段并且对象的 metadata.finalizers 字段包含了值 “foregroundDeletion”，此时该对象依然存在，然后垃圾收集器会删除该对象的所有依赖对象，垃圾收集器在删除了所有“Blocking” 状态的依赖对象（指其子对象中 ownerReference.blockOwnerDeletion=true的对象）之后，然后才会删除对象本身；

DaemonSetController

在 kubernetes 中 daemonset 类似于 linux 上的守护进程会运行在每一个 node 上，在实际场景中，一般会将日志采集或者网络插件采用 daemonset 的方式部署

Statefulset 的基本功能

statefulset 旨在与有状态的应用及分布式系统一起使用，statefulset 中的每个 pod 拥有一个唯一的身份标识，并且所有 pod 名都是按照 {0..N-1} 的顺序进行编号。本文会主要分析 statefulset controller 的设计与实现，在分析源码前先介绍一下 statefulset 的基本使用。

deployment 的功能

deployment 是 kubernetes 中用来部署无状态应用的一个对象，也是最常用的一种对象。
deployment 的本质是控制 replicaSet，replicaSet 会控制 pod，然后由 controller 驱动各个对象达到期望状态。

DeploymentController 是 Deployment 资源的控制器，其通过 DeploymentInformer、ReplicaSetInformer、PodInformer 监听三种资源，当三种资源变化时会触发 DeploymentController 中的 syncLoop 操作。

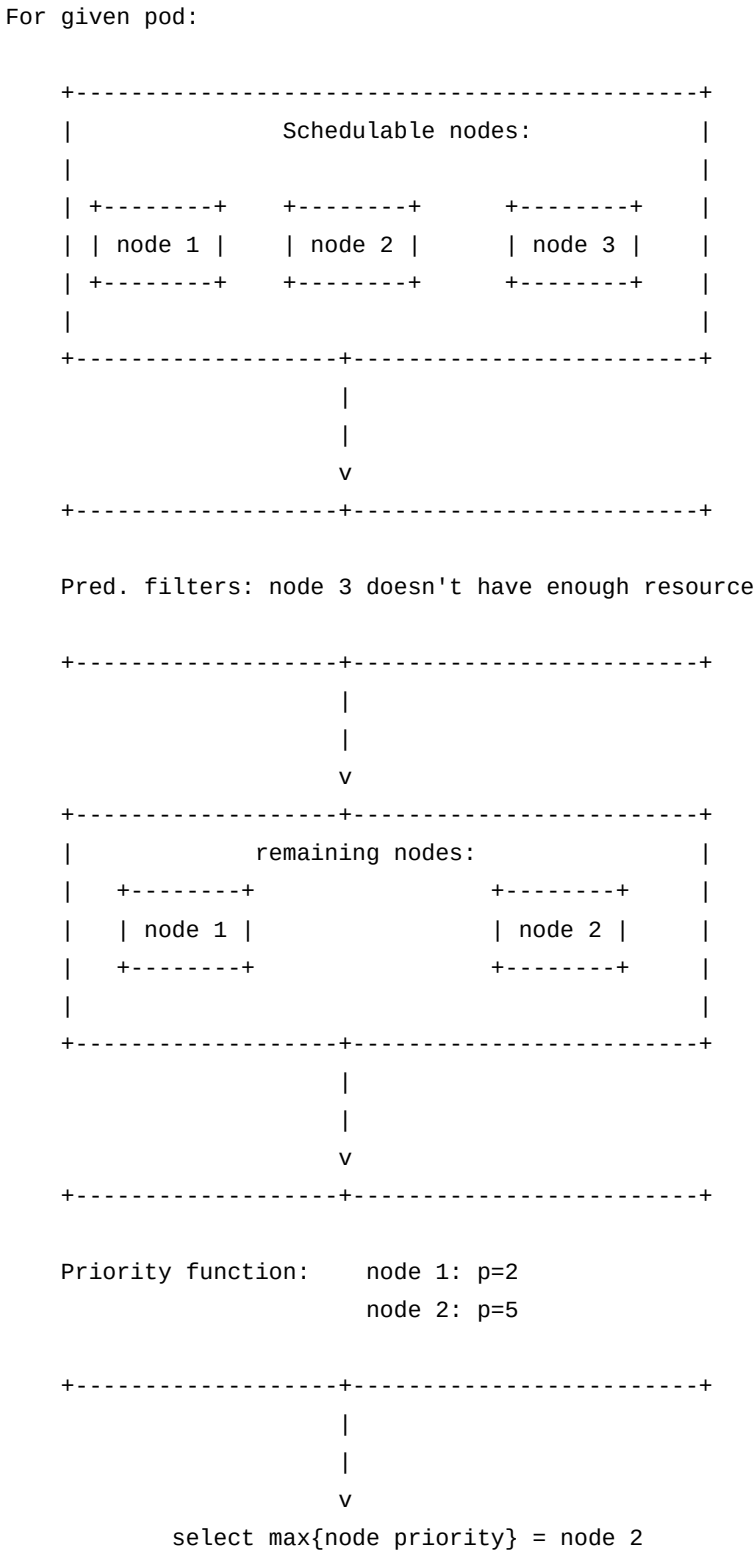
ReplicaSetController

deployment 控制的是 replicaset，而 replicaset 控制 pod 的创建与删除，deployment 通过控制 replicaset 实现了滚动更新、回滚等操作。而 replicaset 会直接控制 pod 的创建与删除，本文会继续从源码层面分析 replicaset 的设计与实现。

在分析源码前先考虑一下 replicaset 的使用场景，在平时的操作中其实我们并不会直接操作 replicaset，replicaset 也仅有几个简单的操作，创建、删除、更新等，但其地位是非常重要的，replicaset 的主要功能就是通过 add/del pod 来达到期望的状态。

kube-scheduler

Kube-scheduler 是 kubernetes 的核心组件之一，也是所有核心组件之间功能比较单一的，其代码也相对容易理解。kube-scheduler 的目的就是为每一个 pod 选择一个合适的 node，整体流程可以概括为三步，获取未调度的 podList，通过执行一系列调度算法为 pod 选择一个合适的 node，提交数据到 apiserver，其核心则是一系列调度算法的设计与执行。



kube-scheduler 目前包含两部分调度算法 predicates 和 priorities，首先执行 predicates 算法过滤部分 node 然后执行 priorities 算法为所有 node 打分，最后从所有 node 中选出分数最高的最为最佳的 node。

kube-proxy

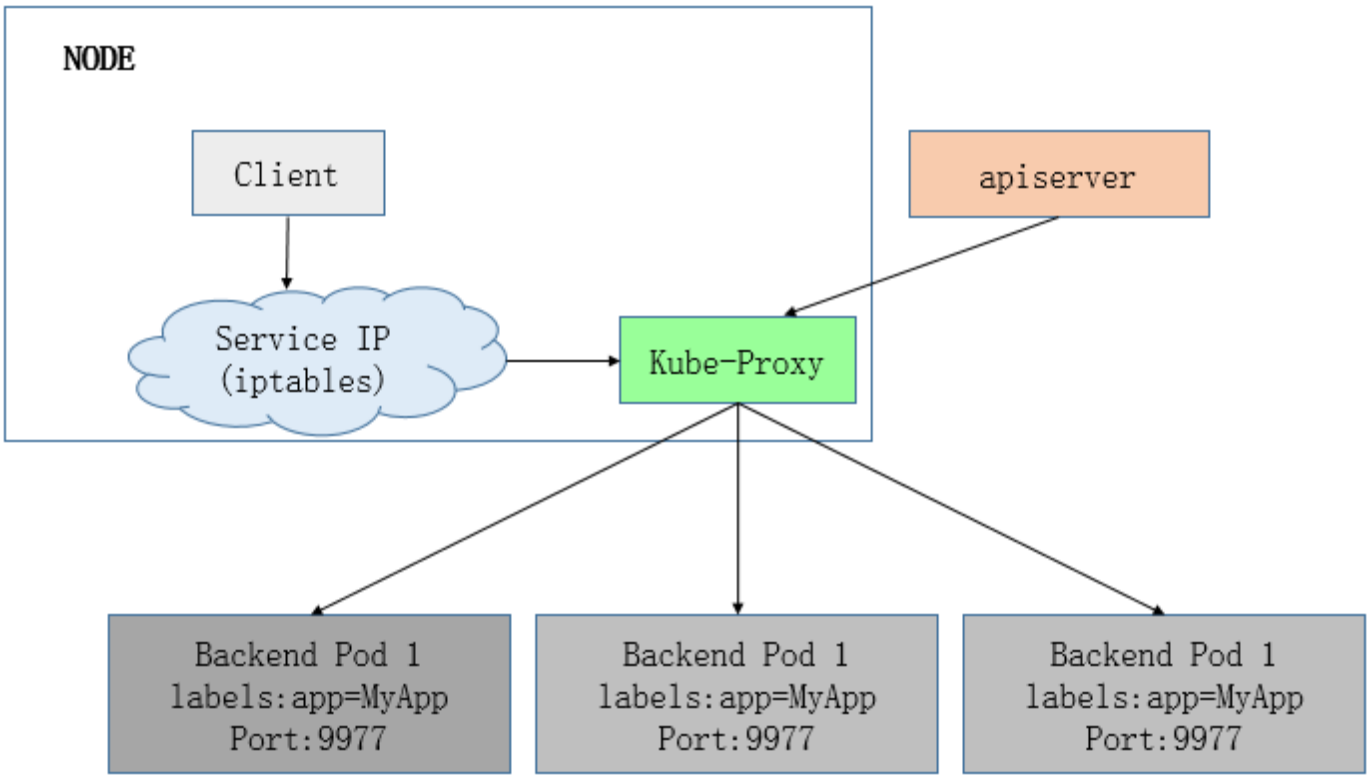
kube-proxy是Kubernetes的核心组件，部署在每个Node节点上，它是实现Kubernetes Service的通信与负载均衡机制的重要组件; kube-proxy负责为Pod创建代理服务，从apiserver获取所有server信息，并根据server信息创建代理服务，实现server到Pod的请求路由和转发，从而实现K8s层级的虚拟转发网络。

简单来说:

- kube-proxy其实就是管理service的访问入口，包括集群内Pod到Service的访问和集群外访问service。
- kube-proxy管理sevice的Endpoints，该service对外暴露一个Virtual IP，也成为Cluster IP, 集群内通过访问这个Cluster IP:Port就能访问到集群内对应的serivce下的Pod。
- service是通过Selector选择的一组Pods的服务抽象，其实就是一个微服务，提供了服务的LB和反向代理的能力，而kube-proxy的主要作用就是负责service的实现。
- service另外一个重要作用是，一个服务后端的Pods可能会随着生存灭亡而发生IP的改变，service的出现，给服务提供了一个固定的IP，而无视后端Endpoint的变化。

kube-proxy 工作原理

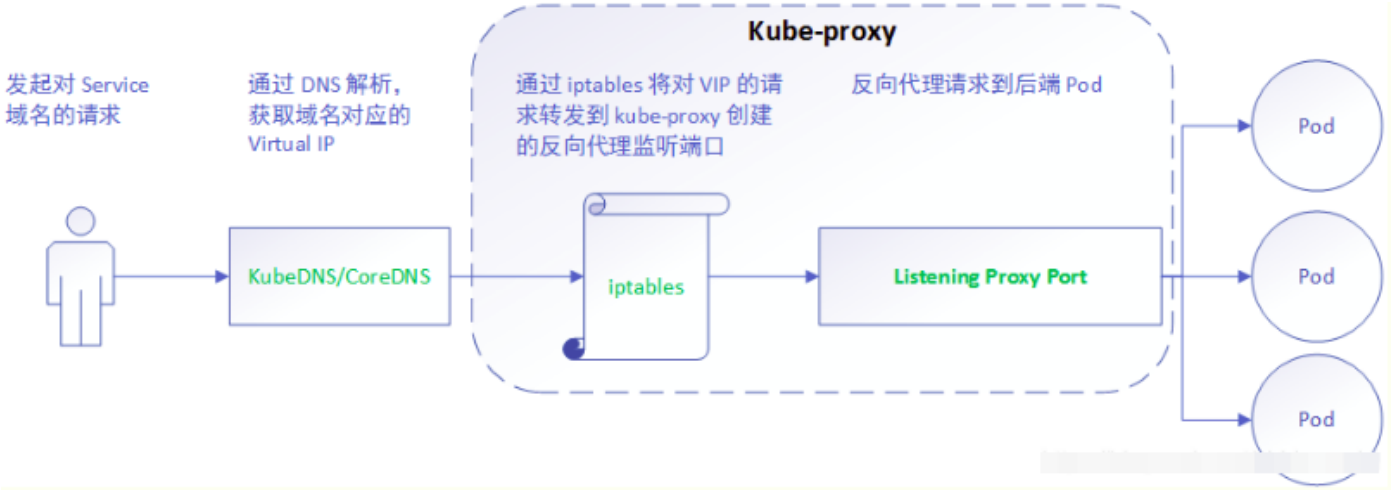
- userspace mode
  userspace是在用户空间，通过kube-proxy来实现service的代理服务, 其原理如下



可见，userspace这种mode最大的问题是，service的请求会先从用户空间进入内核iptables，然后再回到用户空间，由kube-proxy完成后端Endpoints的选择和代理工作，这样流量从用户空间进出

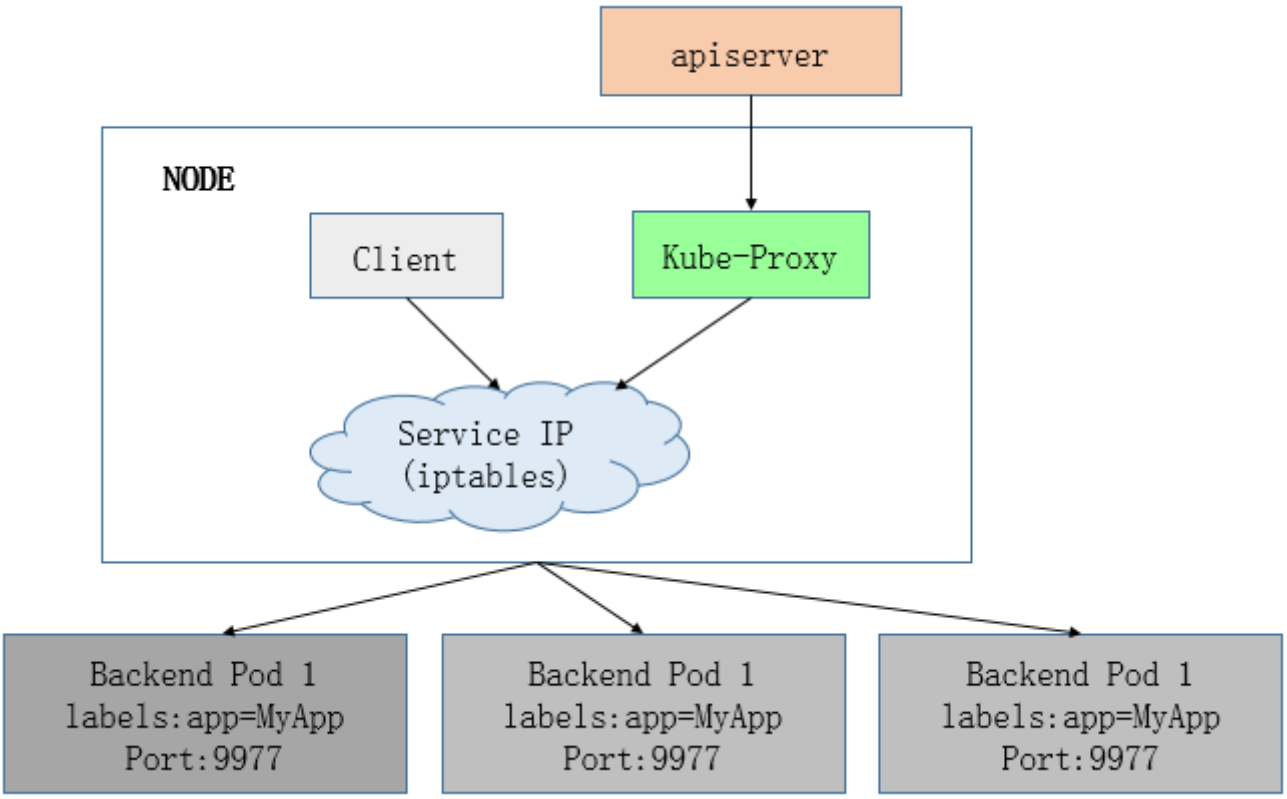
内核带来的性能损耗是不可接受的。这也是k8s v1.0及之前版本中对kube-proxy质疑最大的一点，因此社区就开始研究iptables mode.

userspace这种模式下，kube-proxy 持续监听 Service 以及 Endpoints 对象的变化；对每个 Service，它都为其在本地节点开放一个端口，作为其服务代理端口；发往该端口的请求会采用一定的策略转发给与该服务对应的后端 Pod 实体。kube-proxy 同时会在本地节点设置 iptables 规则，配置一个 Virtual IP，把发往 Virtual IP 的请求重定向到与该 Virtual IP 对应的服务代理端口上。其工作流程大体如下:



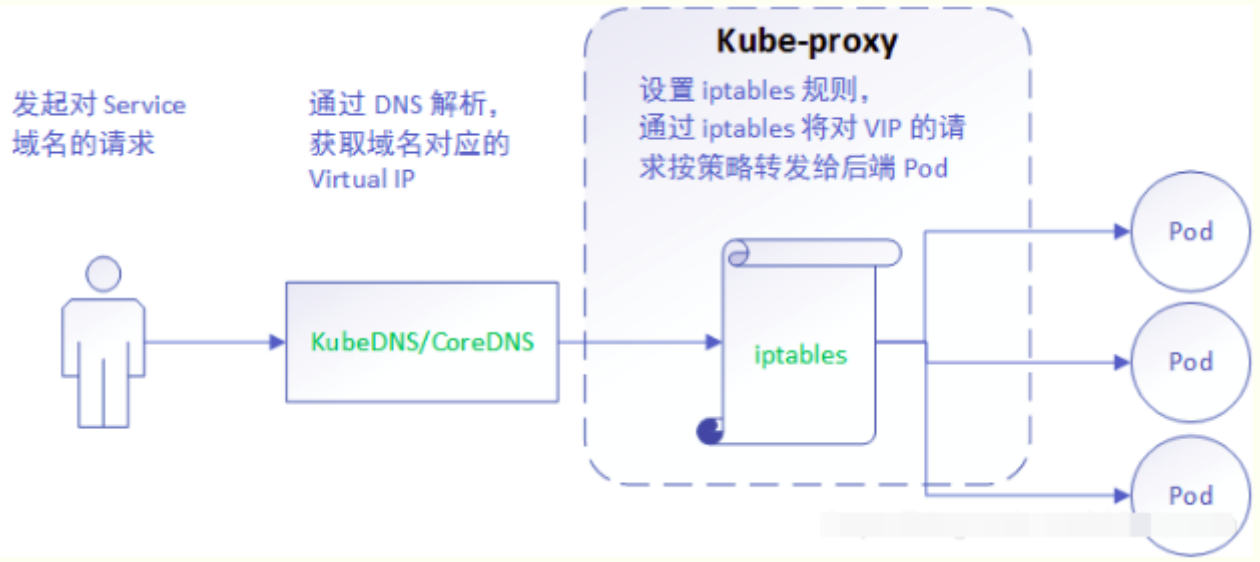
- iptables mode

  该模式完全利用内核iptables来实现service的代理和LB, 这是K8s在v1.2及之后版本默认模式. 工作原理如下:



iptables mode因为使用iptable NAT来完成转发，也存在不可忽视的性能损耗。另外，如果集群中存在上万的Service/Endpoint，那么Node上的iptables rules将会非常庞大，性能还会再打折扣。这也

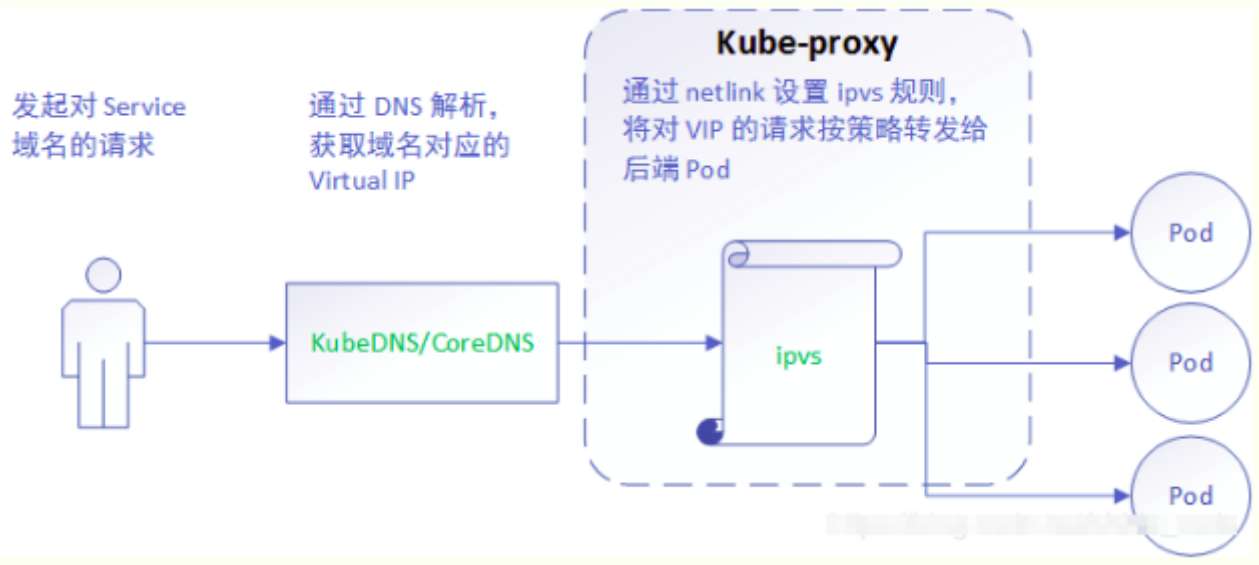
导致目前大部分企业用k8s上生产时，都不会直接用kube-proxy作为服务代理，而是通过自己开发或者通过Ingress Controller来集成HAProxy, Nginx来代替kube-proxy。
iptables 模式与 userspace 相同，kube-proxy 持续监听 Service 以及 Endpoints 对象的变化；但它并不在本地节点开启反向代理服务，而是把反向代理全部交给 iptables 来实现；即 iptables 直接将对 VIP 的请求转发给后端 Pod，通过 iptables 设置转发策略。其工作流程大体如下:



该模式相比 userspace 模式，克服了请求在用户态-内核态反复传递的问题，性能上有所提升，但使用 iptables NAT 来完成转发，存在不可忽视的性能损耗，而且在大规模场景下，iptables 规则的条目会十分巨大，性能上还要再打折扣。

- ipvs mode
  在kubernetes 1.8以上的版本中，对于kube-proxy组件增加了除iptables模式和用户模式之外还支持ipvs模式。kube-proxy ipvs 是基于 NAT 实现的，通过ipvs的NAT模式，对访问k8s service的请求进行虚IP到POD IP的转发。当创建一个 service 后，kubernetes 会在每个节点上创建一个网卡，同时帮你将 Service IP(VIP) 绑定上，此时相当于每个 Node 都是一个 ds，而其他任何 Node 上的 Pod，甚至是宿主机服务(比如 kube-apiserver 的 6443)都可能成为 rs；

与iptables、userspace 模式一样，kube-proxy 依然监听Service以及Endpoints对象的变化, 不过它并不创建反向代理, 也不创建大量的 iptables 规则, 而是通过netlink 创建ipvs规则，并使用k8s Service与Endpoints信息，对所在节点的ipvs规则进行定期同步; netlink 与 iptables 底层都是基于 netfilter 钩子，但是 netlink 由于采用了 hash table 而且直接工作在内核态，在性能上比 iptables 更优。其工作流程大体如下:



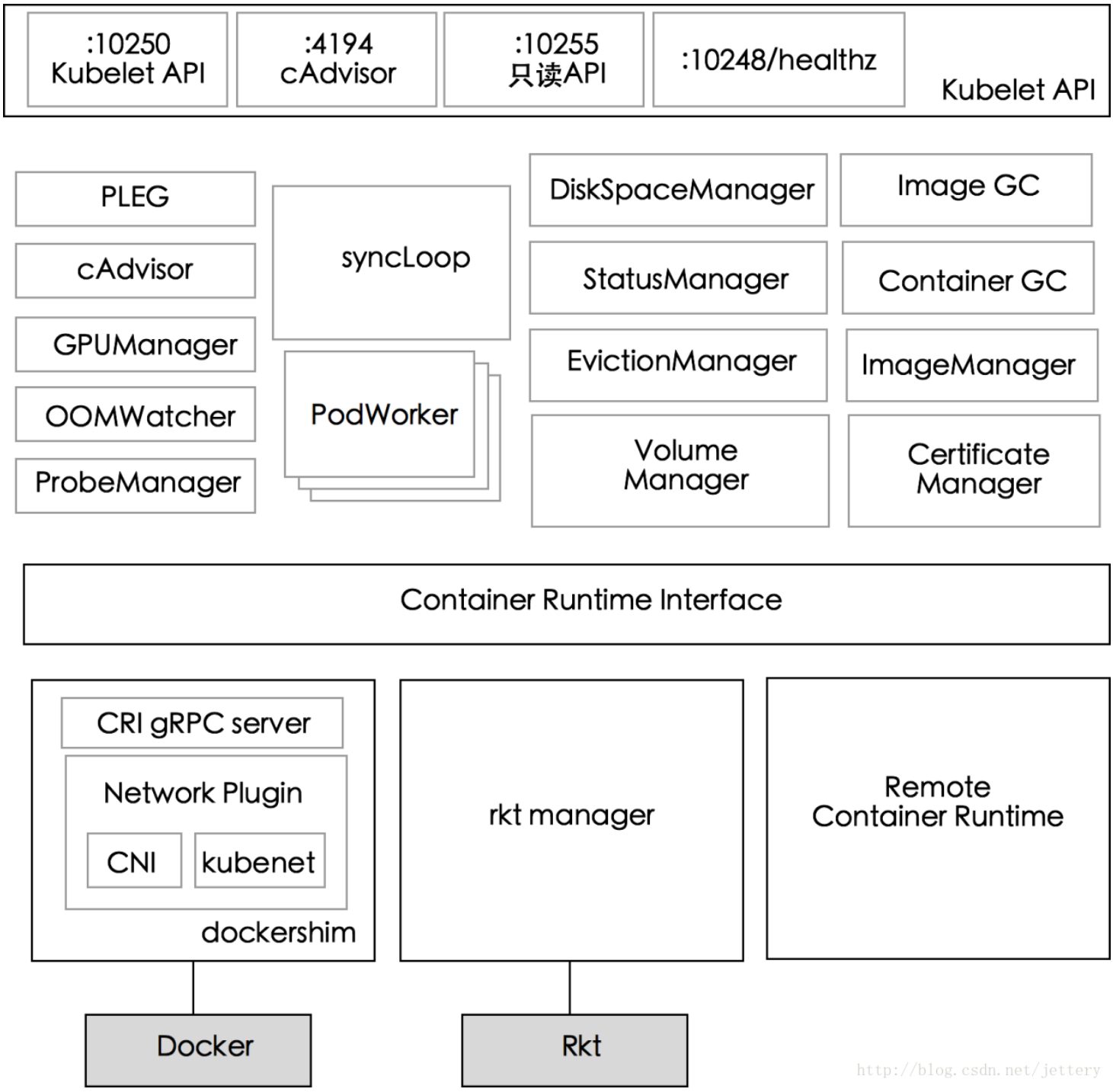
kubelet

kubelet 是运行在每个节点上的主要的“节点代理”，每个节点都会启动 kubelet进程，用来处理 Master 节点下发到本节点的任务，按照 PodSpec 描述来管理Pod 和其中的容器（PodSpec 是用来描述一个 pod 的 YAML 或者 JSON 对象）。

kubelet 通过各种机制（主要通过 apiserver ）获取一组 PodSpec 并保证在这些 PodSpec 中描述的容器健康运行。

kubelet 的主要功能

- pod 管理：kubelet 定期从所监听的数据源获取节点上 pod/container 的期望状态（运行什么容器、运行的副本数量、网络或者存储如何配置等等），并调用对应的容器平台接口达到这个状态。
- 容器健康检查：kubelet 创建了容器之后还要查看容器是否正常运行，如果容器运行出错，就要根据 pod 设置的重启策略进行处理。
- 容器监控：kubelet 会监控所在节点的资源使用情况，并定时向 master 报告，资源使用数据都是通过 cAdvisor 获取的。知道整个集群所有节点的资源情况，对于 pod 的调度和正常运行至关重要。



- 1、PLEG(Pod Lifecycle Event Generator） PLEG 是 kubelet 的核心模块,PLEG 会一直调用 container runtime 获取本节点 containers/sandboxes 的信息，并与自身维护的 pods cache 信息进行对比，生成对应的 PodLifecycleEvent，然后输出到 eventChannel 中，通过 eventChannel 发送到 kubelet syncLoop 进行消费，然后由 kubelet syncPod 来触发 pod 同步处理过程，最终达到用户的期望状态。
- 2、cAdvisor cAdvisor（https://github.com/google/cadvisor）是 google 开发的容器监控工具，集成在 kubelet 中，起到收集本节点和容器的监控信息，大部分公司对容器的监控数据都是从 cAdvisor 中获取的 ，cAvisor 模块对外提供了 interface 接口，该接口也被 imageManager，OOMWatcher，containerManager 等所使用。

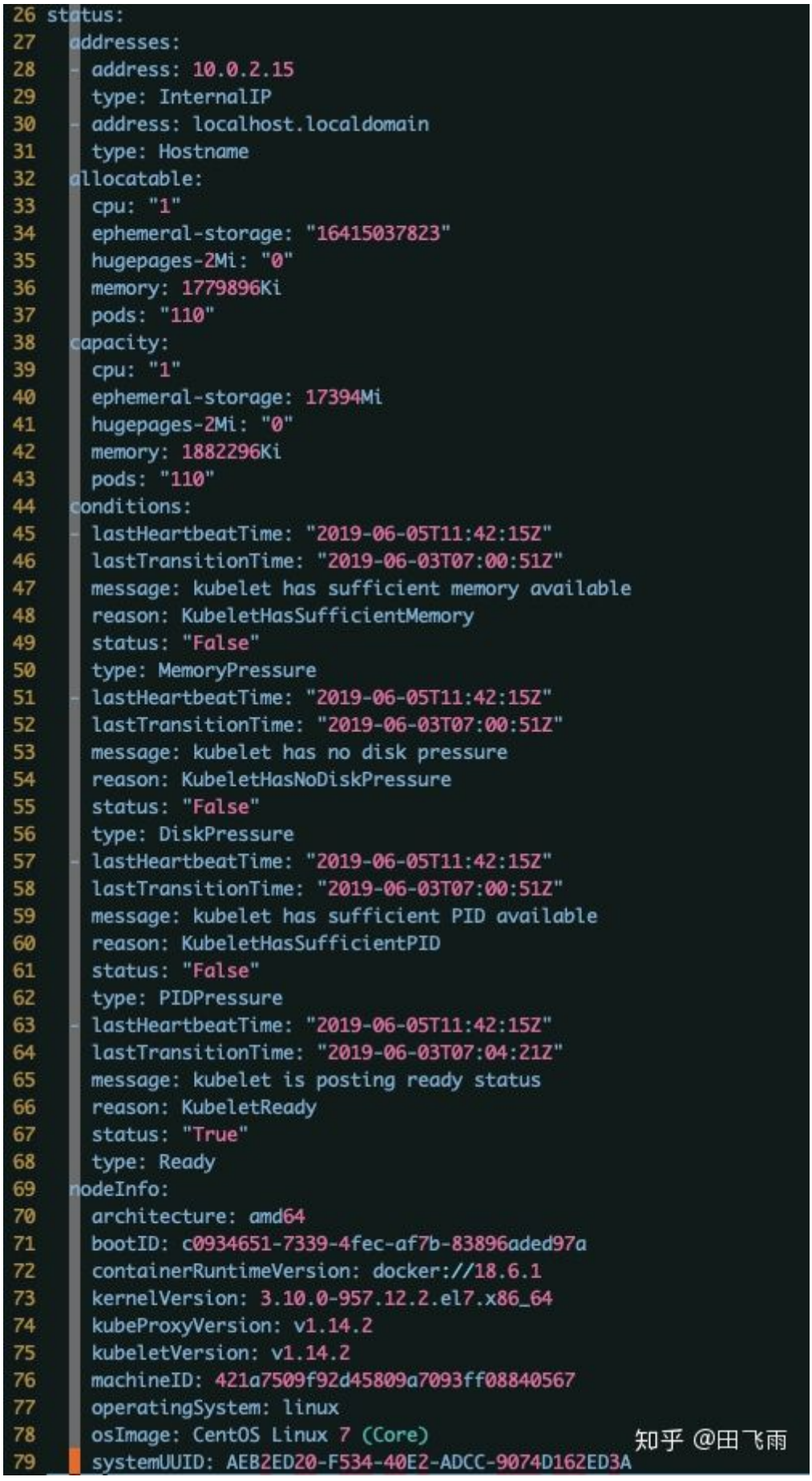
- 3、OOMWatcher 系统 OOM 的监听器，会与 cadvisor 模块之间建立 SystemOOM,通过 Watch方式从 cadvisor 那里收到的 OOM 信号，并产生相关事件。
- 4、probeManager probeManager 依赖于 statusManager,livenessManager,containerRefManager，会定时去监控 pod 中容器的健康状况，当前支持两种类型的探针：livenessProbe 和readinessProbe。 livenessProbe：用于判断容器是否存活，如果探测失败，kubelet 会 kill 掉该容器，并根据容器的重启策略做相应的处理。readinessProbe：用于判断容器是否启动完成，将探测成功的容器加入到该 pod 所在 service 的 endpoints 中，反之则移除。readinessProbe 和 livenessProbe 有三种实现方式：http、tcp 以及 cmd。
- 5、statusManager statusManager 负责维护状态信息，并把 pod 状态更新到 apiserver，但是它并不负责监控 pod 状态的变化，而是提供对应的接口供其他组件调用，比如 probeManager。
- 6、containerRefManager 容器引用的管理，相对简单的Manager，用来报告容器的创建，失败等事件，通过定义 map 来实现了 containerID 与 v1.ObjectReferece 容器引用的映射。
- 7、evictionManager 当节点的内存、磁盘或 inode 等资源不足时，达到了配置的 evict 策略， node 会变为 pressure 状态，此时 kubelet 会按照 qosClass 顺序来驱赶 pod，以此来保证节点的稳定性。可以通过配置 kubelet 启动参数 --eviction-hard= 来决定 evict 的策略值。
- 8、imageGC imageGC 负责 node 节点的镜像回收，当本地的存放镜像的本地磁盘空间达到某阈值的时候，会触发镜像的回收，删除掉不被 pod 所使用的镜像，回收镜像的阈值可以通过 kubelet 的启动参数 --image-gc-high-threshold 和 --image-gc-low-threshold 来设置。
- 9、containerGC containerGC 负责清理 node 节点上已消亡的 container，具体的 GC 操作由 runtime 来实现。
- 10、imageManager 调用 kubecontainer 提供的 PullImage/GetImageRef/ListImages/RemoveImage/ImageStates 方法来保证pod 运行所需要的镜像。
- 11、volumeManager 负责 node 节点上 pod 所使用 volume 的管理，volume 与 pod 的生命周期关联，负责 pod 创建删除过程中 volume 的 mount/umount/attach/detach 流程，kubernetes 采用 volume Plugins 的方式，实现存储卷的挂载等操作，内置几十种存储插件。
- 12、containerManager 负责 node 节点上运行的容器的 cgroup 配置信息，kubelet 启动参数如果指定 --cgroups-per-qos 的时候，kubelet 会启动 goroutine 来周期性的更新 pod 的 cgroup 信息，维护其正确性，该参数默认为 true，实现了 pod 的Guaranteed/BestEffort/Burstable 三种级别的 Qos。
- 13、runtimeManager containerRuntime 负责 kubelet 与不同的 runtime 实现进行对接，实现对于底层 container 的操作，初始化之后得到的 runtime 实例将会被之前描述的组件所使用。可以通过 kubelet 的启动参数 --container-runtime 来定义是使用docker 还是 rkt，默认是 docker。
- 14、podManager podManager 提供了接口来存储和访问 pod 的信息，维持 static pod 和 mirror pods 的关系，podManager 会被statusManager/volumeManager/runtimeManager 所调用，podManager 的接口处理流程里面会调用 secretManager 以及 configMapManager。

kubelet 上报哪些状态

- Addresses

HostName：Hostname 。可以通过 kubelet 的 --hostname-override 参数进行覆盖。
ExternalIP：通常是可以外部路由的 node IP 地址（从集群外可访问）。
InternalIP：通常是仅可在集群内部路由的 node IP 地址。

- Condition


status:
addresses:
- address: 10.0.2.15
type: InternalIP
- address: localhost.localdomain
type: Hostname
allocatable:
cpu: "1"
ephemeral-storage: "16415037823"
hugepages-2Mi: "0"
memory: 1779896Ki
pods: "110"
capacity:
cpu: "1"
ephemeral-storage: 17394Mi
hugepages-2Mi: "0"
memory: 1882296Ki
pods: "110"
conditions:
- lastHeartbeatTime: "2019-06-05T11:42:15Z"
lastTransitionTime: "2019-06-03T07:00:51Z"
message: kubelet has sufficient memory available
reason: KubeletHasSufficientMemory
status: "False"
type: MemoryPressure
- lastHeartbeatTime: "2019-06-05T11:42:15Z"
lastTransitionTime: "2019-06-03T07:00:51Z"
message: kubelet has no disk pressure
reason: KubeletHasNoDiskPressure
status: "False"
type: DiskPressure
- lastHeartbeatTime: "2019-06-05T11:42:15Z"
lastTransitionTime: "2019-06-03T07:00:51Z"
message: kubelet has sufficient PID available
reason: KubeletHasSufficientPID
status: "False"
type: PIDPressure
- lastHeartbeatTime: "2019-06-05T11:42:15Z"
lastTransitionTime: "2019-06-03T07:04:21Z"
message: kubelet is posting ready status
reason: KubeletReady
status: "True"
type: Ready
nodeInfo:
architecture: amd64
bootID: c0934651-7339-4fec-af7b-83896aded97a
containerRuntimeVersion: docker://18.6.1
kernelVersion: 3.10.0-957.12.2.el7.x86_64
kubeProxyVersion: v1.14.2
kubeletVersion: v1.14.2
machineID: 421a7509f92d45809a7093ff08840567
operatingSystem: linux
osImage: CentOS Linux 7 (Core)
systemUUID: AEB2ED20-F534-40E2-ADCC-9074D162ED3A
知乎 @田飞雨

- Capacity

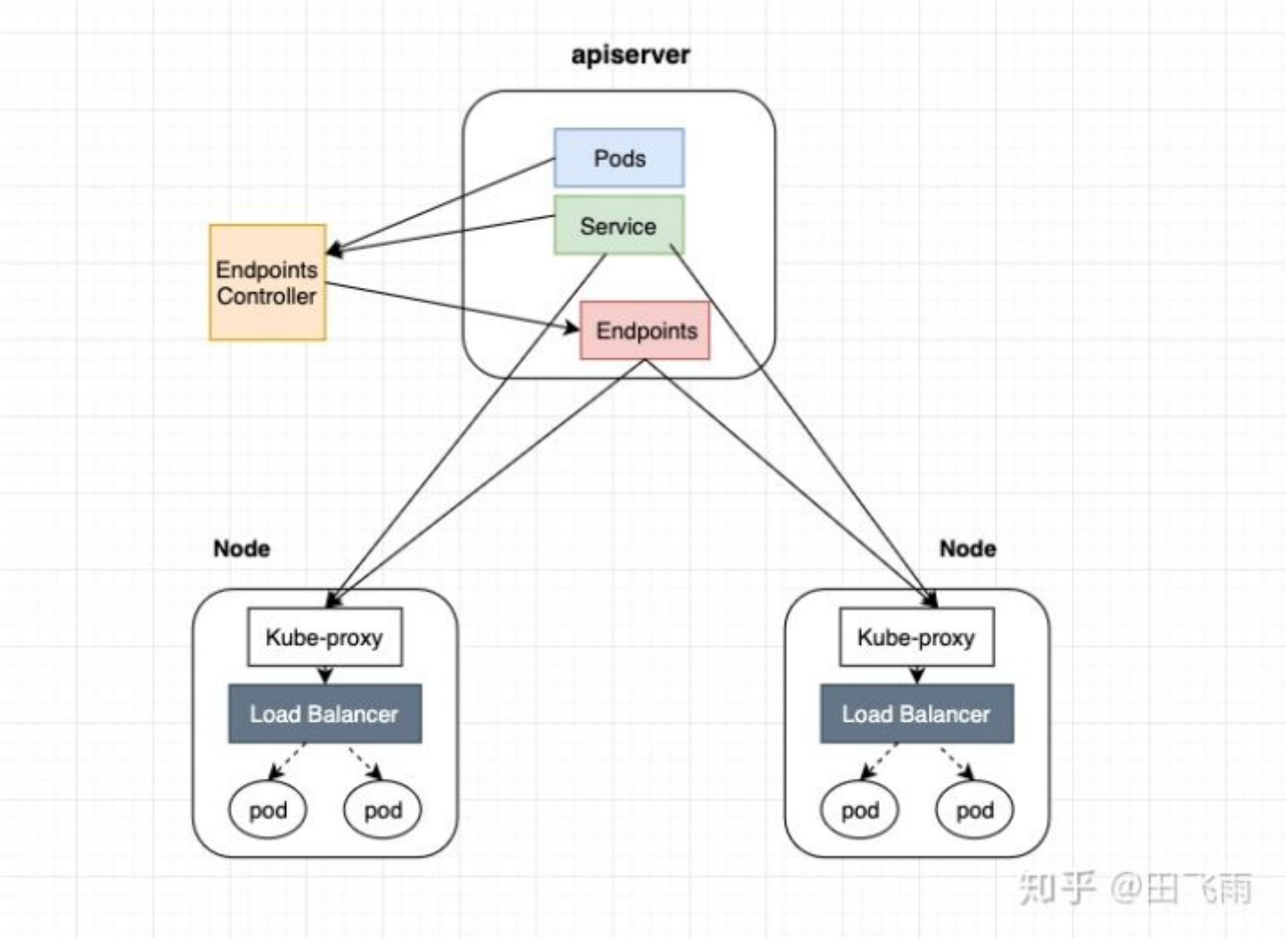
描述 node 上的可用资源：CPU、内存和可以调度到该 node 上的最大 pod 数量。

- Info

描述 node 的一些通用信息，例如内核版本、Kubernetes 版本（kubelet 和 kube-proxy 版本）、Docker 版本 （如果使用了）

Service

在 kubernetes 中，当创建带有多个副本的 deployment 时，kubernetes 会创建出多个 pod，此时即一个服务后端有多个容器，那么在 kubernetes 中负载均衡怎么做，容器漂移后 ip 也会发生变化，如何做服务发现以及会话保持？这就是 service 的作用，service 是一组具有相同 label pod 集合的抽象，集群内外的各个服务可以通过 service 进行互相通信，当创建一个 service 对象时也会对应创建一个 endpoint 对象，endpoint 是用来做容器发现的，service 只是将多个 pod 进行关联，实际的路由转发都是由 kubernetes 中的 kube-proxy 组件来实现，因此，service 必须结合 kube-proxy 使用，kube-proxy 组件可以运行在 kubernetes 集群中的每一个节点上也可以只运行在单独的几个节点上，其会根据 service 和 endpoints 的变动来改变节点上 iptables 或者 ipvs 中保存的路由规则。



endpoints controller 是负责生成和维护所有 endpoints 对象的控制器，监听 service 和对应 pod 的变化，更新对应 service 的 endpoints 对象。当用户创建 service 后 endpoints controller 会监听 pod 的状态，当 pod 处于 running 且准备就绪时，endpoints controller 会将 pod ip 记录到 endpoints 对象中，因此，service 的容器发现是通过 endpoints 来实现的。而 kube-proxy 会监听 service 和 endpoints 的更新并调用其代理模块在主机上刷新路由转发规则。

service 的负载均衡

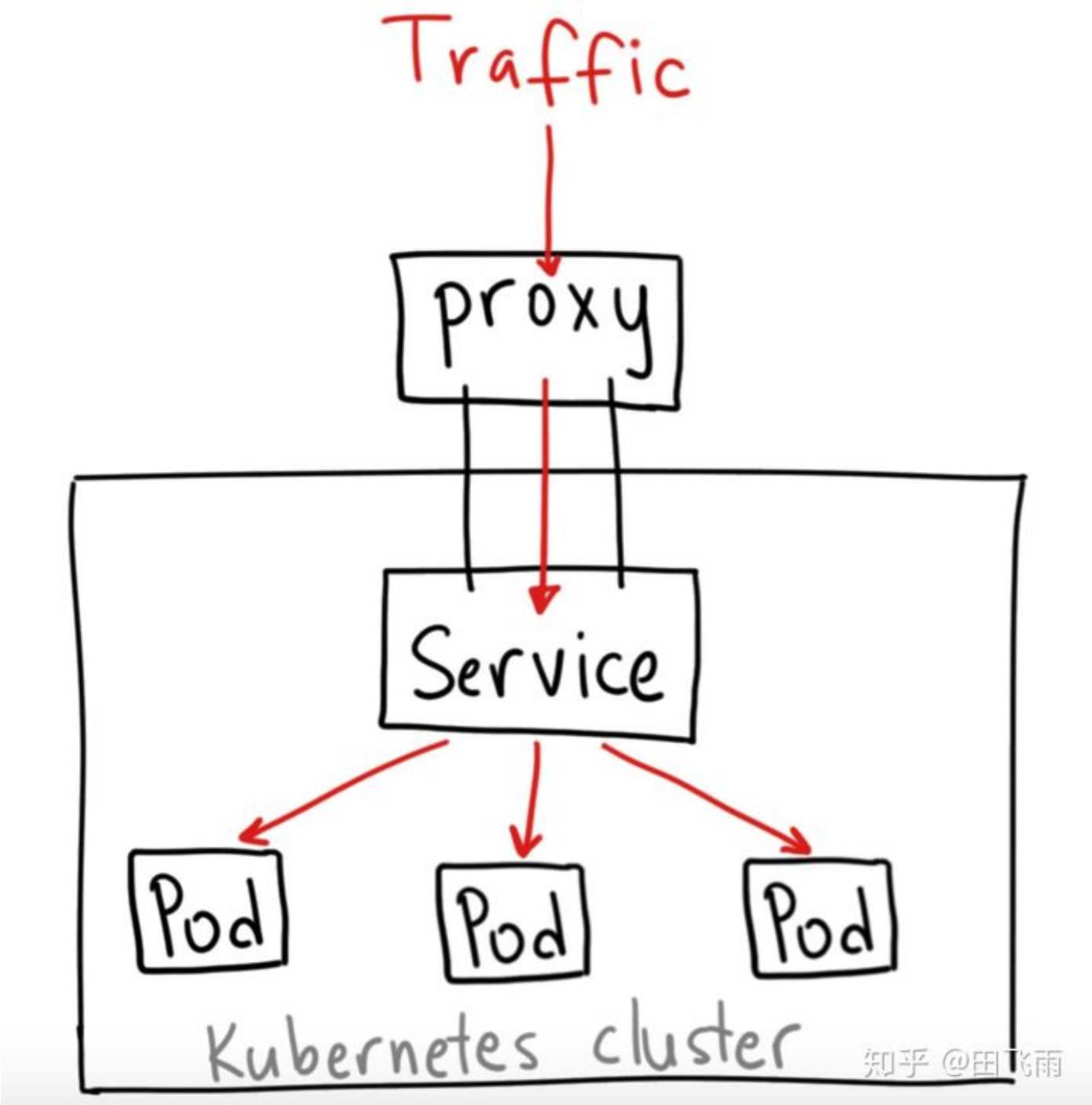
- userspace 模式
- iptables 模式
- ipvs 模式

service 的类型

service 支持的类型也就是 kubernetes 中服务暴露的方式，默认有四种 ClusterIP、NodePort、LoadBalancer、ExternelName，此外还有 Ingress，下面会详细介绍每种类型 service 的具体使用场

景。

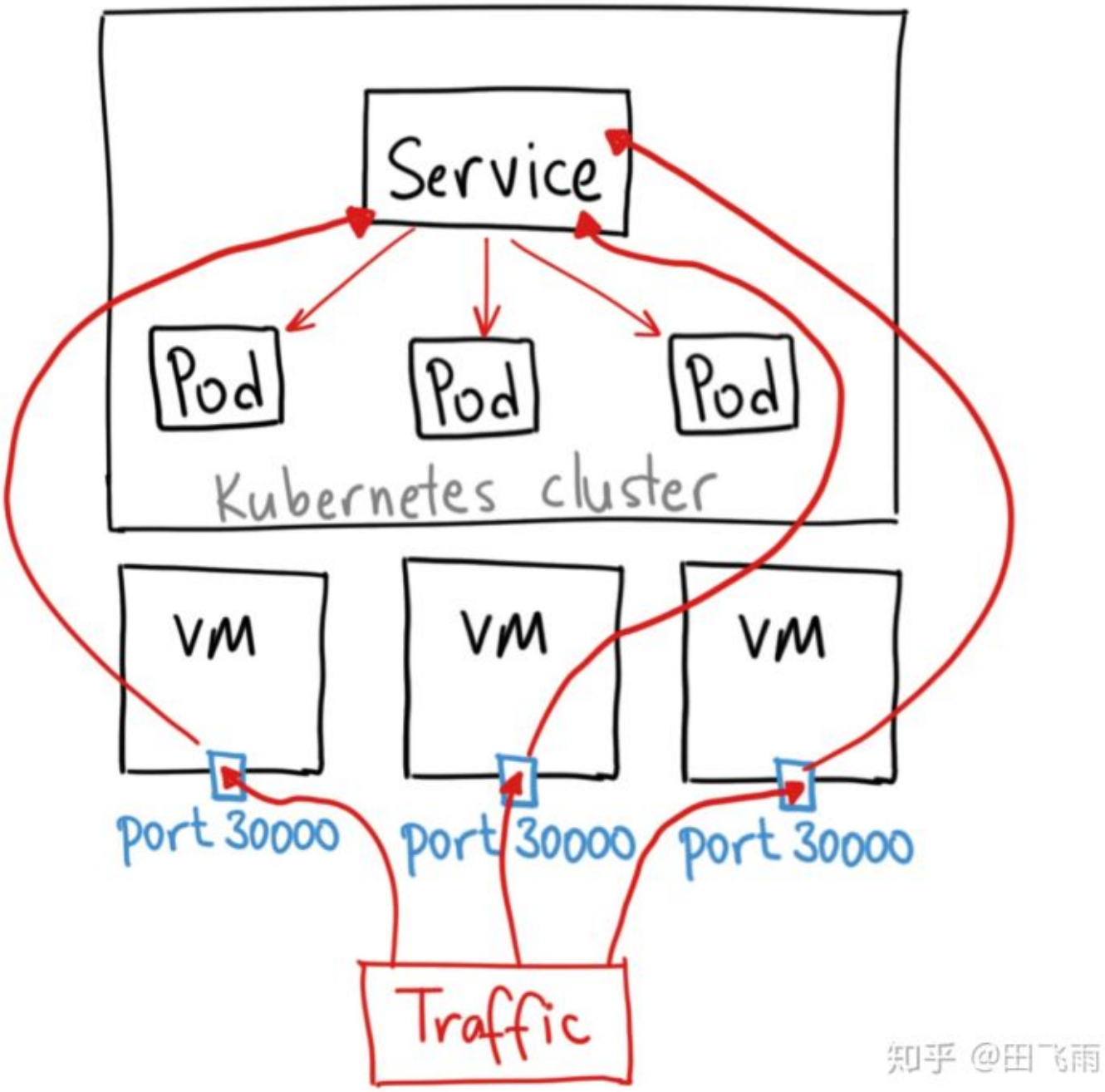
- ClusterIP
  ClusterIP 类型的 service 是 kubernetes 集群默认的服务暴露方式，它只能用于集群内部通信，可以被各 pod 访问，其访问方式为：
  > pod ---> ClusterIP:ServicePort --> (iptables)DNAT --> PodIP:containePort



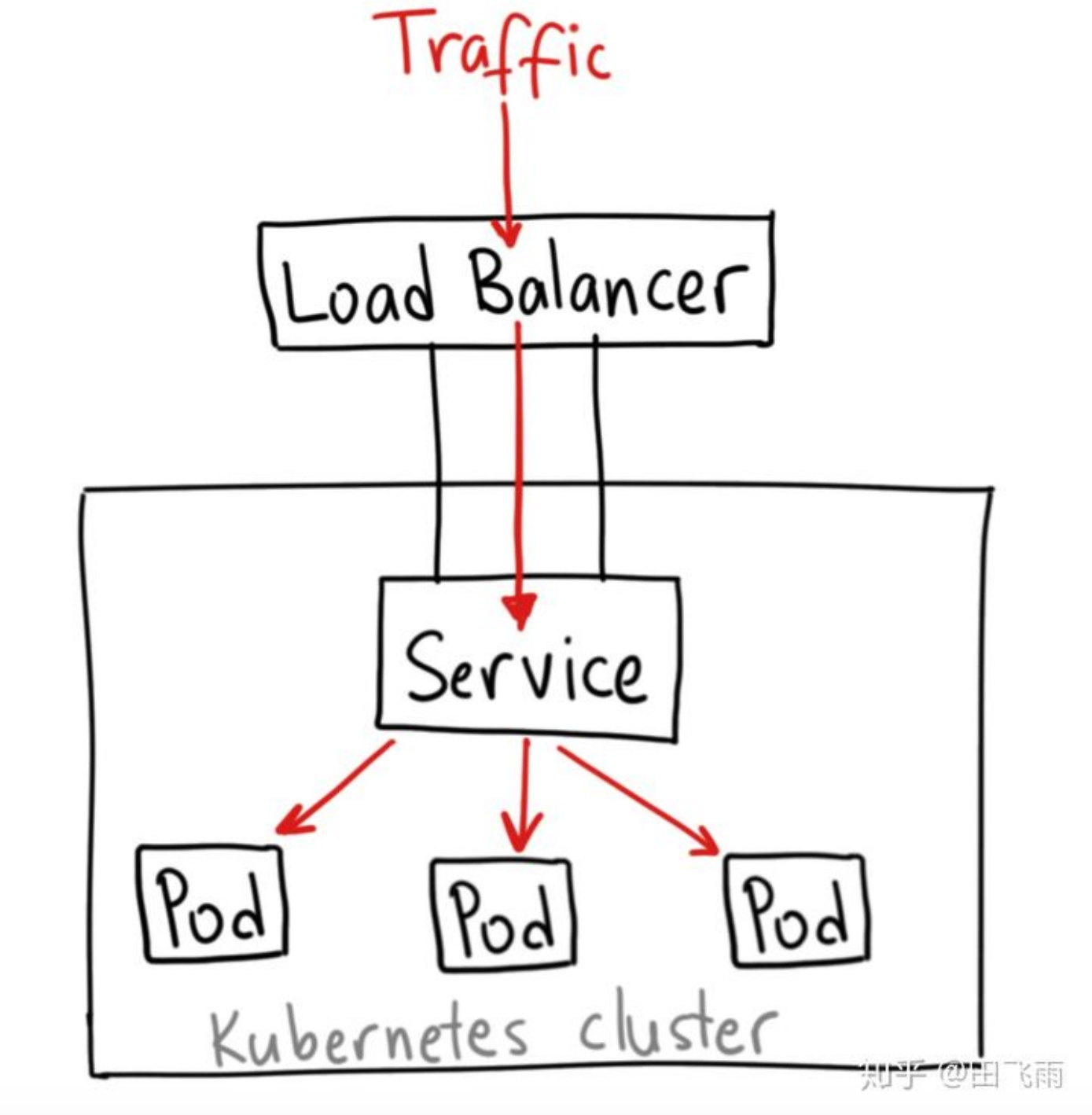
- NodePort
  如果你想要在集群外访问集群内部的服务，可以使用这种类型的 service，NodePort 类型的 service 会在集群内部署了 kube-proxy 的节点打开一个指定的端口，之后所有的流量直接发送到这

个端口，然后会被转发到 service 后端真实的服务进行访问。Nodeport 构建在 ClusterIP 上，其访问链路如下所示：

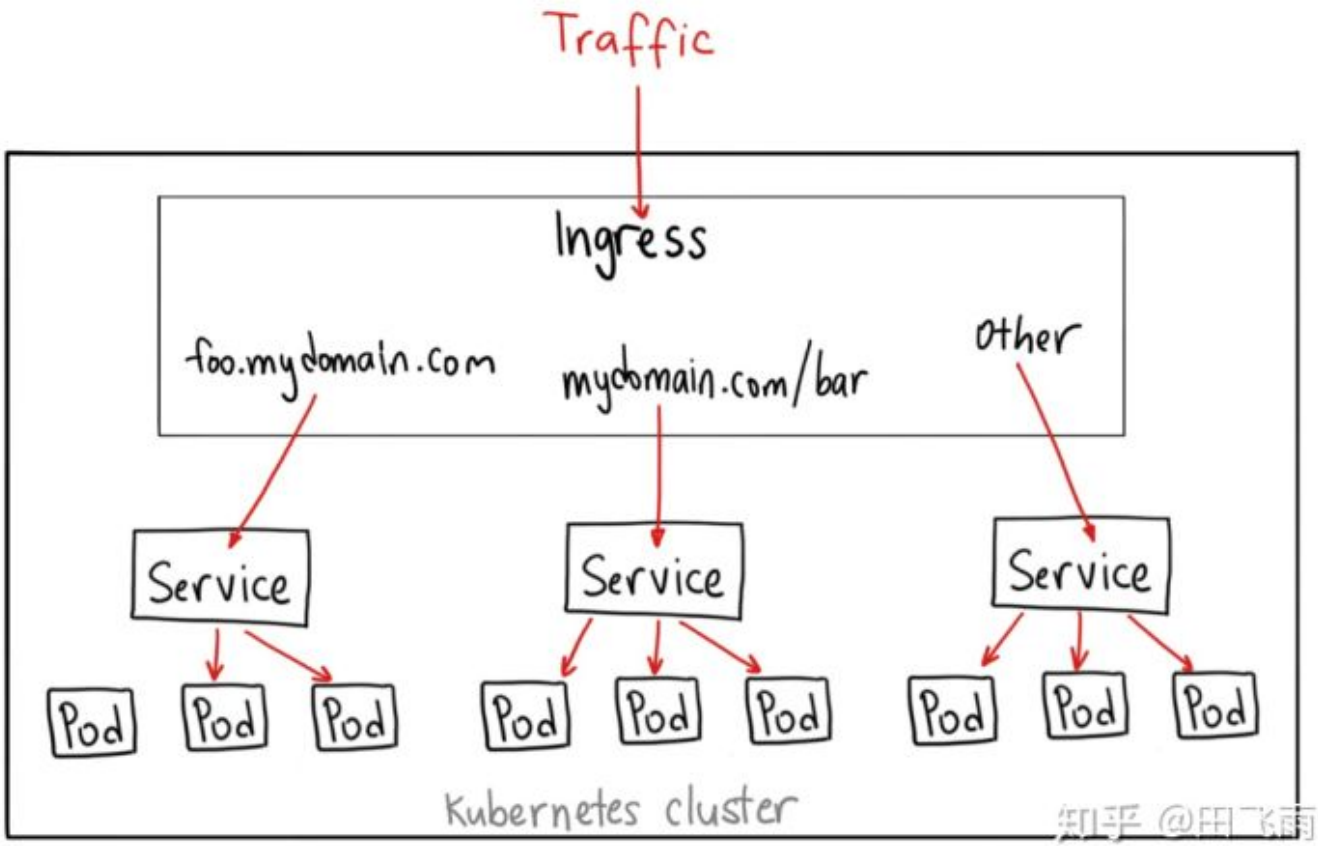
> client ---> NodeIP:NodePort ---> ClusterIP:ServicePort ---> (iptables)DNAT ---> PodIP:containePort



- LoadBalancer
  LoadBalancer 类型的 service 通常和云厂商的 LB 结合一起使用，用于将集群内部的服务暴露到外网，云厂商的 LoadBalancer 会给用户分配一个 IP，之后通过该 IP 的流量会转发到你的 service 上。



- Ingress
  Ingress 其实不是 service 的一个类型，但是它可以作用于多个 service，被称为 service 的 service，作为集群内部服务的入口，Ingress 作用在七层，可以根据不同的 url，将请求转发到不同的 service 上。



- ExternelName
  通过 CNAME 将 service 与 externalName 的值(比如：http://foo.bar.example.com)映射起来，这种方式用的比较少。

service 的服务发现

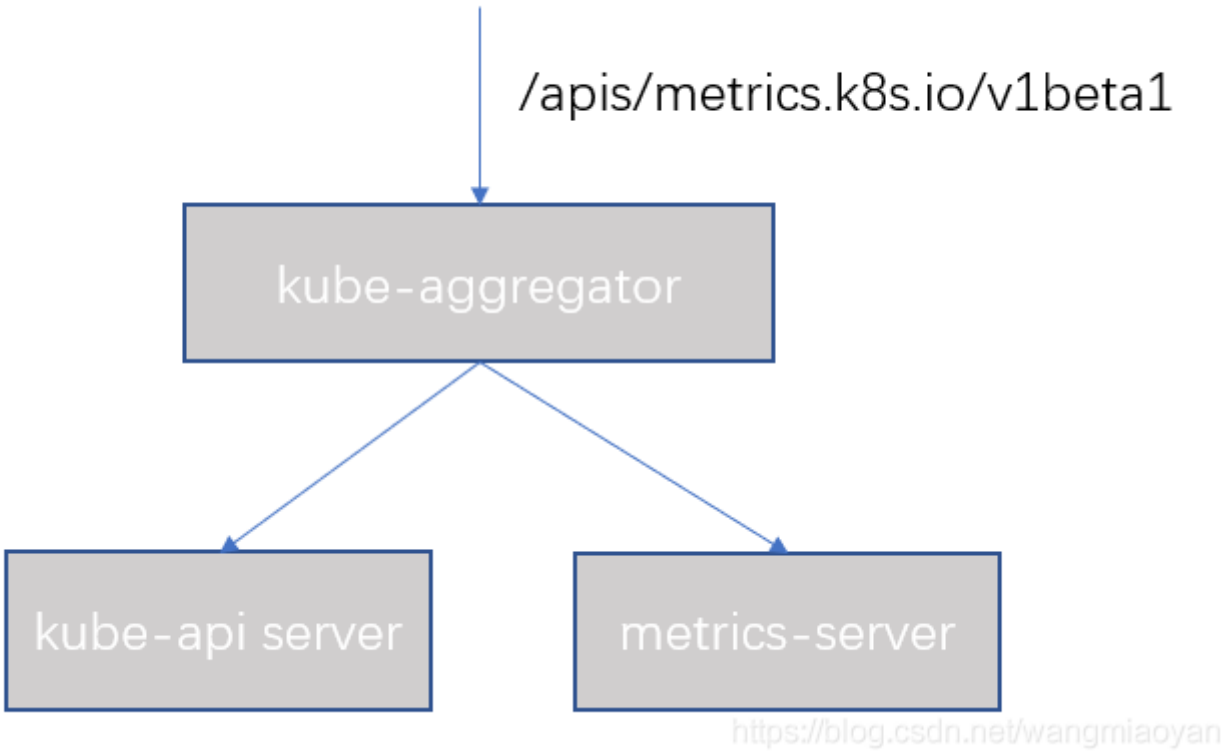
- 环境变量
- DNS

从kubectl top看K8S监控原理

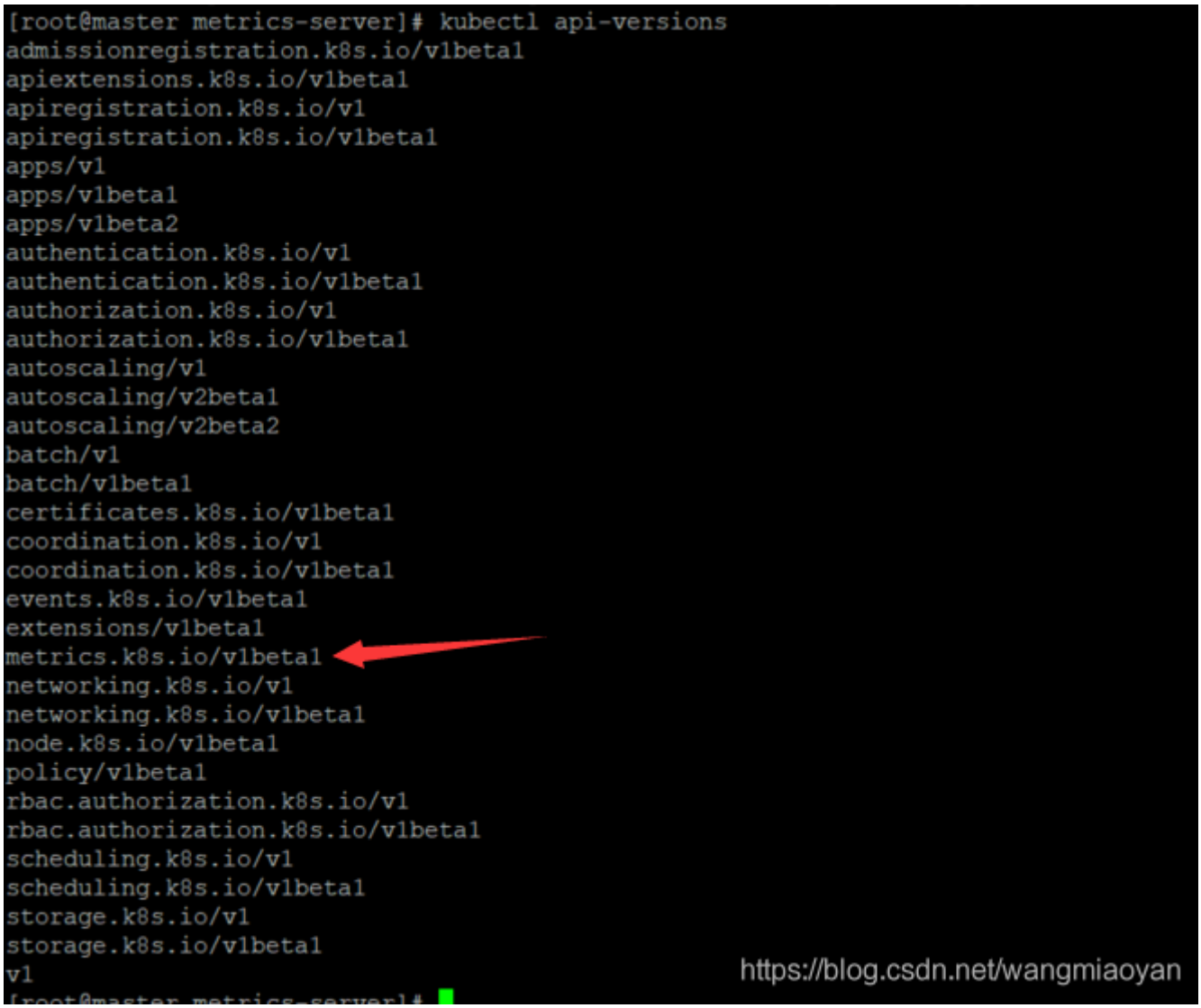
kubectl top 可以很方便地查看node、pod的实时资源使用情况：如CPU、内存。这篇文章会介绍其数据链路和实现原理，同时借kubectl top 阐述 k8s 中的监控体系，窥一斑而知全豹

kubectl top 是基础命令，但是需要部署配套的组件才能获取到监控值

- 1.8以下：部署 heapter
- 1.8以上：部署 metric-server

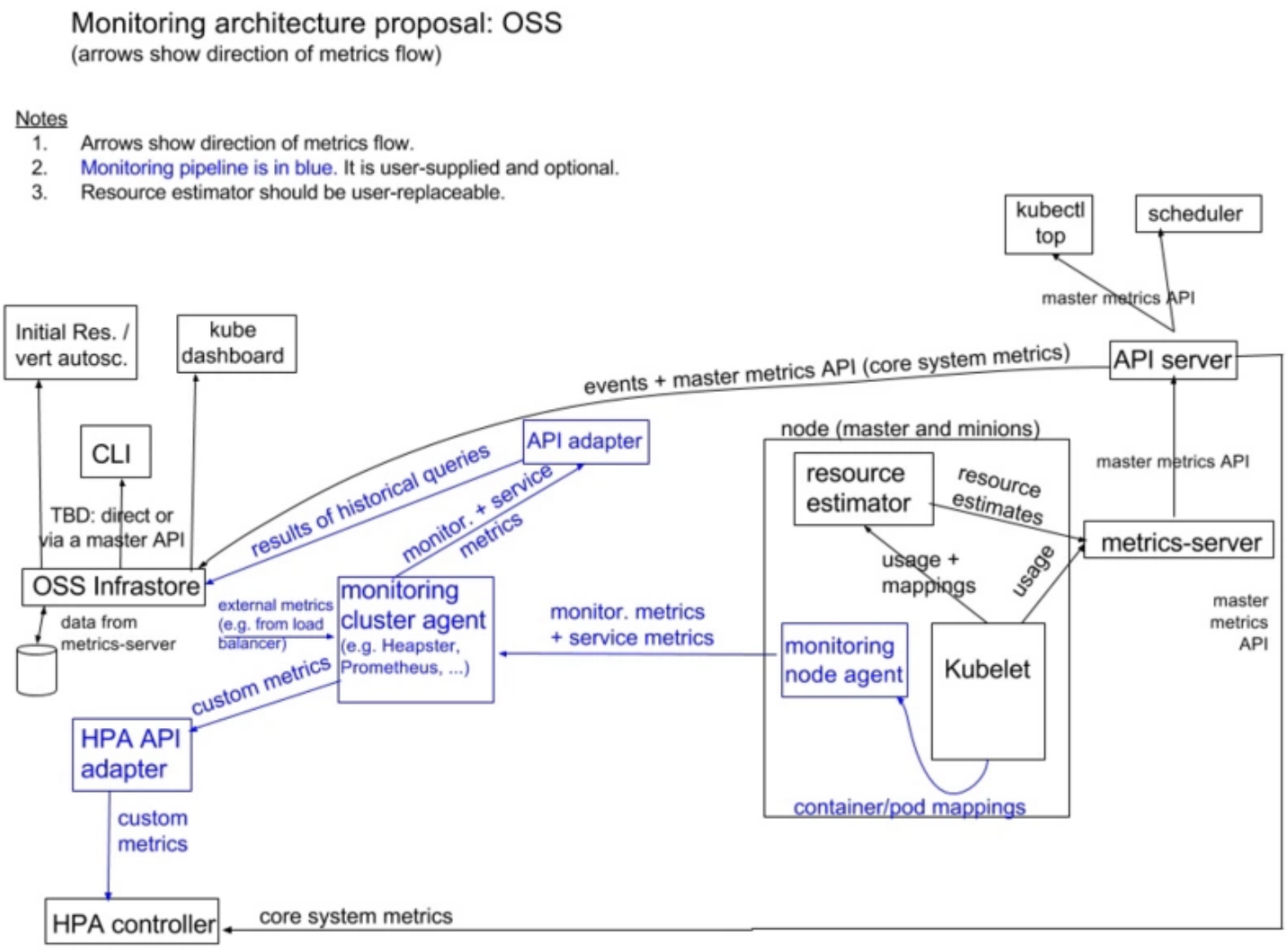


布署完成后, API里增加了metrics 接口



Metrics server定时从Kubelet的Summary API(类似/ap1/v1/nodes/nodename/stats/summary)采集指标信息，这些聚合过的数据将存储在内存中，且以metric-api的形式暴露出去。

Metrics server复用了api-server的库来实现自己的功能，比如鉴权、版本等，为了实现将数据存放在内存中吗，去掉了默认的etcd存储，引入了内存存储（即实现Storage interface)。因为存放在内存中，因此监控数据是没有持久化的，可以通过第三方存储来拓展。



Metrics server出现后，新的Kubernetes 监控架构将变成上图的样子

- 核心流程（黑色部分）：这是 Kubernetes正常工作所需要的核心度量，从 Kubelet、cAdvisor 等获取度量数据，再由metrics-server提供给 Dashboard、HPA 控制器等使用。
- 监控流程（蓝色部分）：基于核心度量构建的监控流程，比如 Prometheus 可以从 metrics-server 获取核心度量，从其他数据源（如 Node Exporter 等）获取非核心度量，再基于它们构建监控告警系统。