

# Kubernetes高阶

## 设计和实现

# etcd

- Etcd是CoreOS基于Raft开发的分布式key-value存储，可用于服务发现、共享配置以及一致性保障（如数据库选主、分布式锁等）。

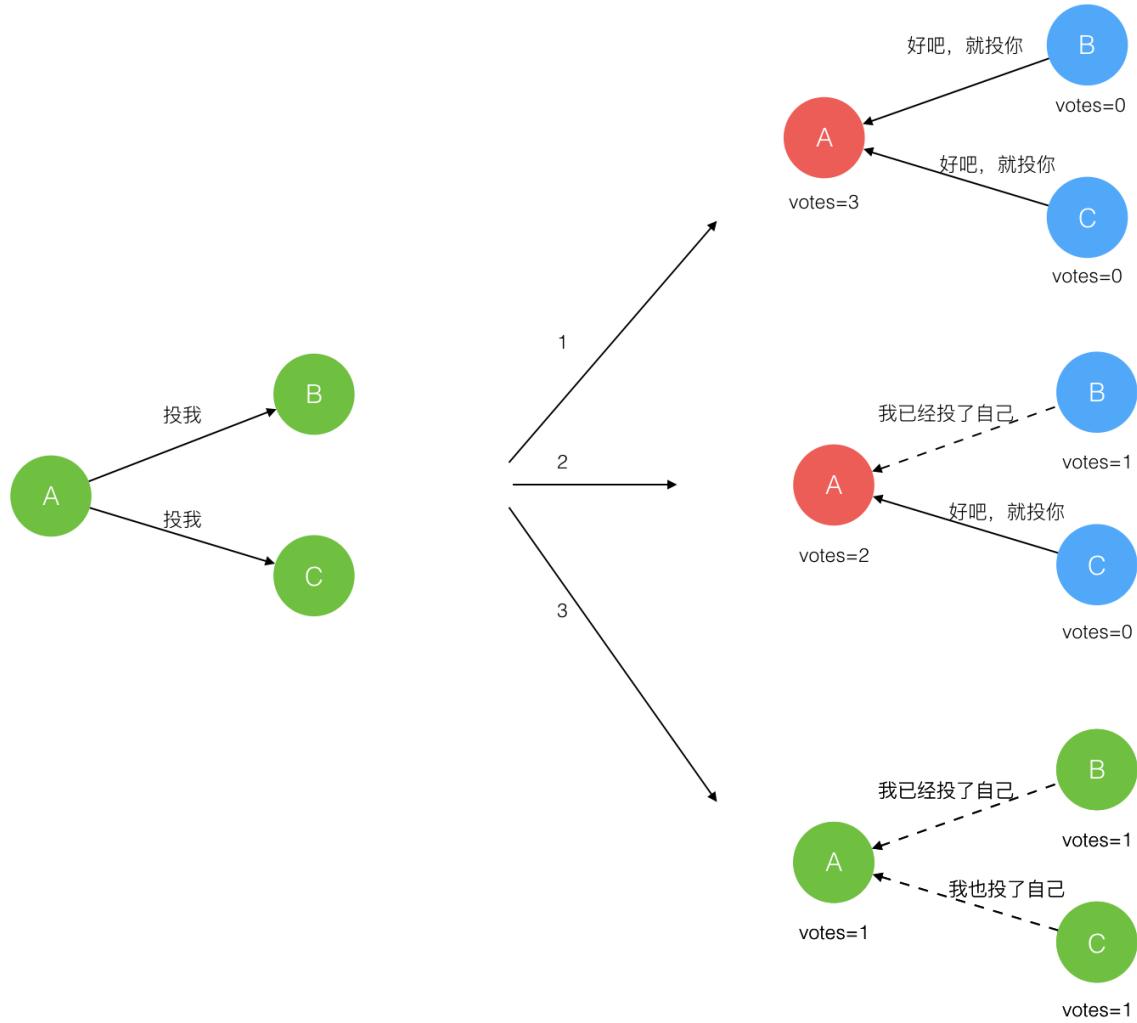
# 主要功能

- 基本的key-value存储
- 监听机制
- key的过期及续约机制，用于监控和服务发现
- 原子CAS和CAD，用于分布式锁和leader选举

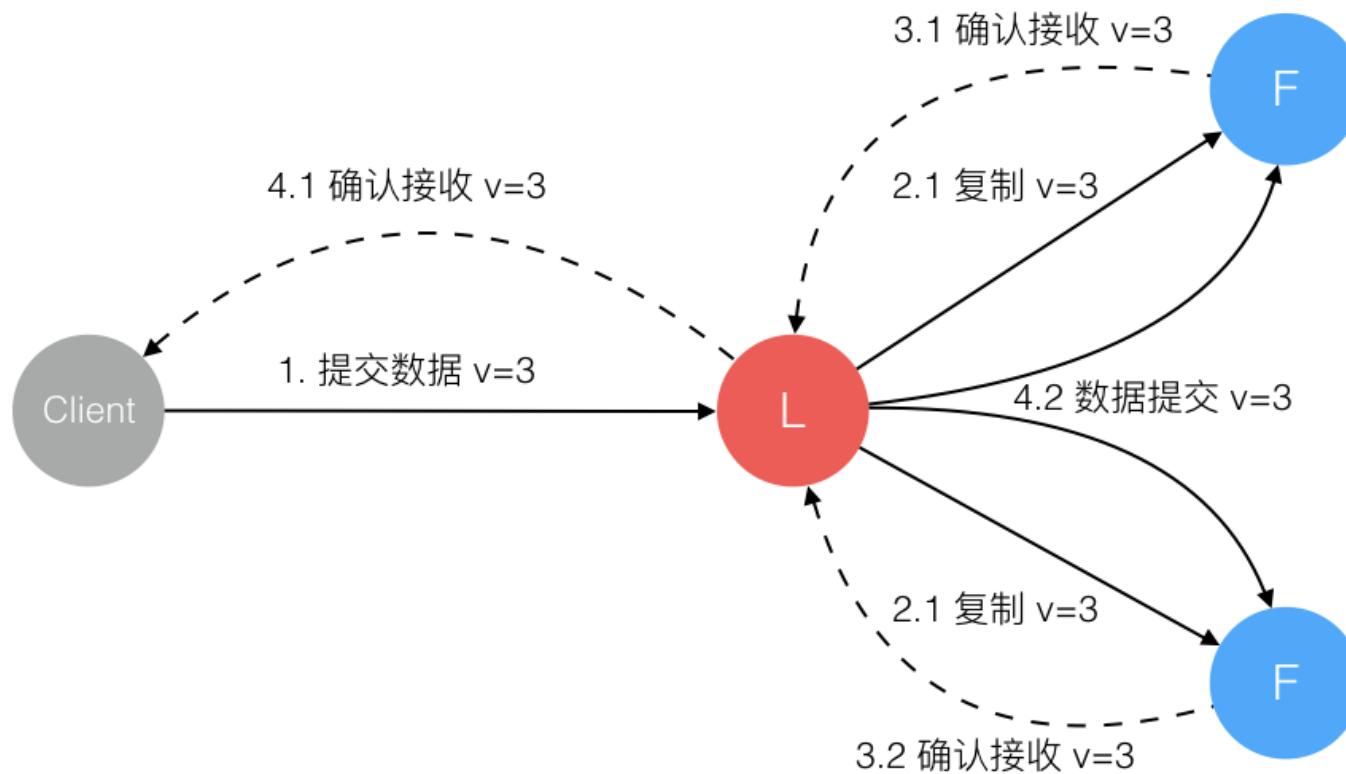
# Etcdb 基于RAFT的一致性

- 选举方法
  - 1) 初始启动时，节点处于 follower 状态并被设定一个 election timeout，如果在这一时间周期内没有收到来自 leader 的 heartbeat，节点将发起选举：将自己切换为 candidate 之后，向集群中其它 follower 节点发送请求，询问其是否选举自己成为 leader。
  - 2) 当收到来自集群中过半数节点的接受投票后，节点即成为 leader，开始接收保存 client 的数据并向其它的 follower 节点同步日志。如果没有达成一致，则 candidate 随机选择一个等待间隔（150ms ~ 300ms）再次发起投票，得到集群中半数以上 follower 接受的 candidate 将成为 leader
  - 3) leader 节点依靠定时向 follower 发送 heartbeat 来保持其地位。
  - 4) 任何时候如果其它 follower 在 election timeout 期间都没有收到来自 leader 的 heartbeat，同样会将自己的状态切换为 candidate 并发起选举。每成功选举一次，新 leader 的任期（Term）都会比之前 leader 的任期大1。

# 选举



# 一致性



# 日志复制

- 当接Leader收到客户端的日志（事务请求）后先把该日志追加到本地的Log中，然后通过heartbeat把该Entry同步给其他Follower， Follower接收到日志后记录日志然后向Leader发送ACK，当Leader收到大多数 ( $n/2+1$ ) Follower的ACK信息后将该日志设置为已提交并追加到本地磁盘中，通知客户端并在下个heartbeat中Leader将通知所有的Follower将该日志存储在自己的本地磁盘中。
-

# 安全性

- 安全性是用于保证每个节点都执行相同序列的安全机制，如当某个Follower在当前Leader commit Log时变得不可用了，稍后可能该Follower又会被选举为Leader，这时新Leader可能会用新的Log覆盖先前已committed的Log，这就是导致节点执行不同序列；Safety就是用于保证选举出来的Leader一定包含先前 committed Log的机制；
- 选举安全性（Election Safety）：每个任期（Term）只能选举出一个Leader
- Leader完整性（Leader Completeness）：指Leader日志的完整性，当Log在任期Term1被Commit后，那么以后任期Term2、Term3…等的Leader必须包含该Log；Raft在选举阶段就使用Term的判断用于保证完整性：当请求投票的该Candidate的Term较大或Term相同Index更大则投票，否则拒绝该请求。

# 失效处理

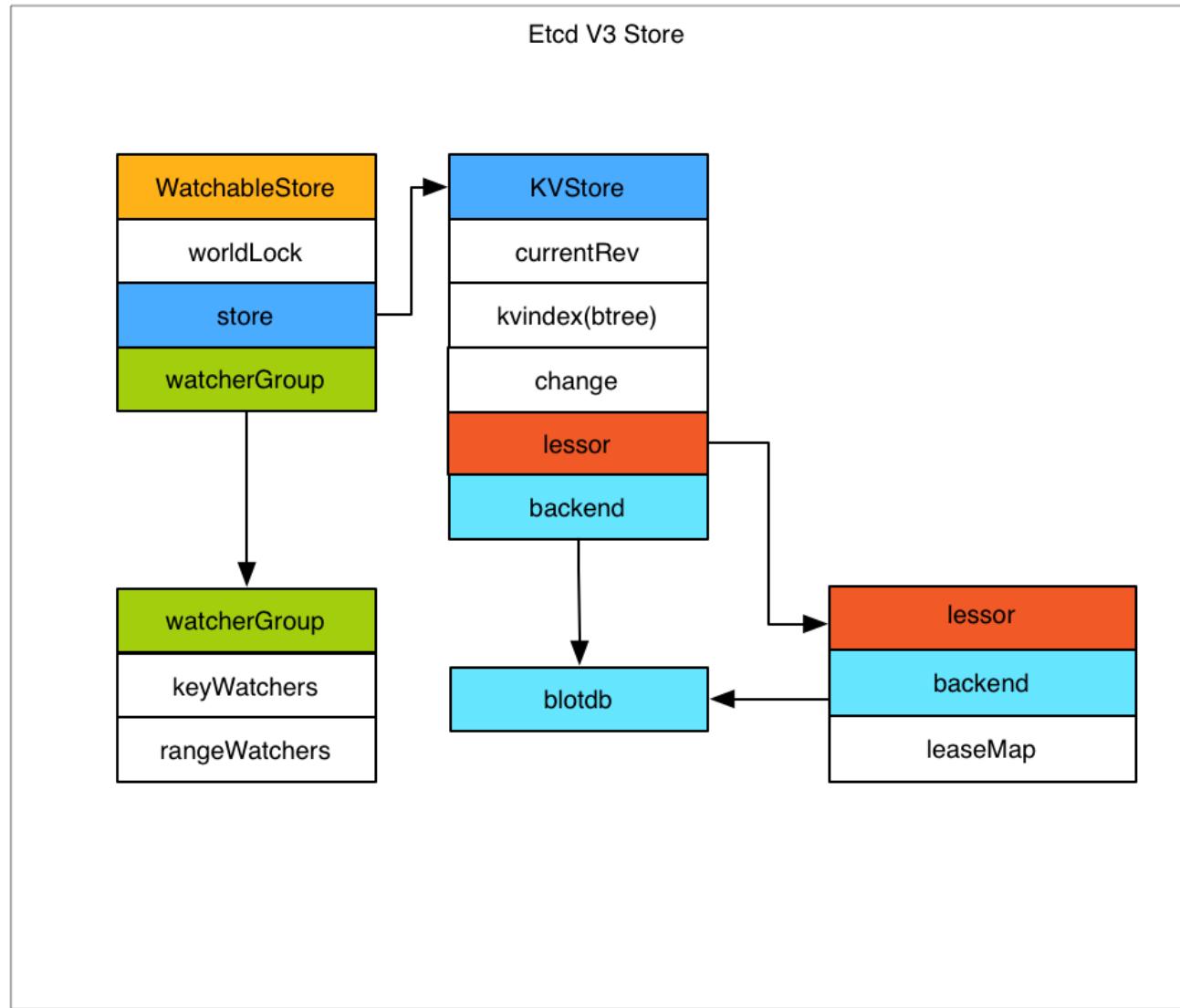
- 1) Leader失效：其他没有收到heartbeat的节点会发起新的选举，而当Leader恢复后由于步进数小会自动成为follower（日志也会被新leader的日志覆盖）
- 2) follower节点不可用：follower 节点不可用的情况相对容易解决。因为集群中的日志内容始终是从 leader 节点同步的，只要这一节点再次加入集群时重新从 leader 节点处复制日志即可。
- 3) 多个candidate：冲突后candidate将随机选择一个等待间隔（150ms ~ 300ms）再次发起投票，得到集群中半数以上follower接受的candidate将成为leader

# wal日志

- wal日志是二进制的，解析出来后是以上数据结构LogEntry。其中第一个字段type，只有两种，一种是0表示Normal，1表示ConfChange（ConfChange表示Etcd本身的配置变更同步，比如有新的节点加入等）。第二个字段是term，每个term代表一个主节点的任期，每次主节点变更term就会变化。第三个字段是index，这个序号是严格有序递增的，代表变更序号。第四个字段是二进制的data，将raft request对象的pb结构整个保存下。Etcd源码下有个tools/etc-dump-logs，可以将wal日志dump成文本查看，可以协助分析raft协议。
- raft协议本身不关心应用数据，也就是data中的部分，一致性都通过同步wal日志来实现，每个节点将从主节点收到的data apply到本地的存储，raft只关心日志的同步状态，如果本地存储实现的有bug，比如没有正确的将data apply到本地，也可能会导致数据不一致。

Entry			
type	term	index	data

# Etcd v3 存储，Watch以及过期机制



# 存储机制

- Etcd v3 store 分为两部分，一部分是内存中的索引，kvindex，是基于google开源的一个golang的btree实现的，另外一部分是后端存储。按照它的设计，backend可以对接多种存储，当前使用的boltdb。boltdb是一个单机的支持事务的kv存储，Etcd 的事务是基于boltdb的事务实现的。Etcd 在boltdb中存储的key是reversion，value是 Etcd 自己的key-value组合，也就是说 Etcd 会在boltdb中把每个版本都保存下，从而实现了多版本机制。
- reversion主要由两部分组成，第一部分main rev，每次事务进行加一，第二部分sub rev，同一个事务中的每次操作加一。
- Etcd 提供了命令和设置选项来控制compact，同时支持put操作的参数来精确控制某个key的历史版本数。
- 内存kvindex保存的就是key和reversion之前的映射关系，用来加速查询。

# Watch机制

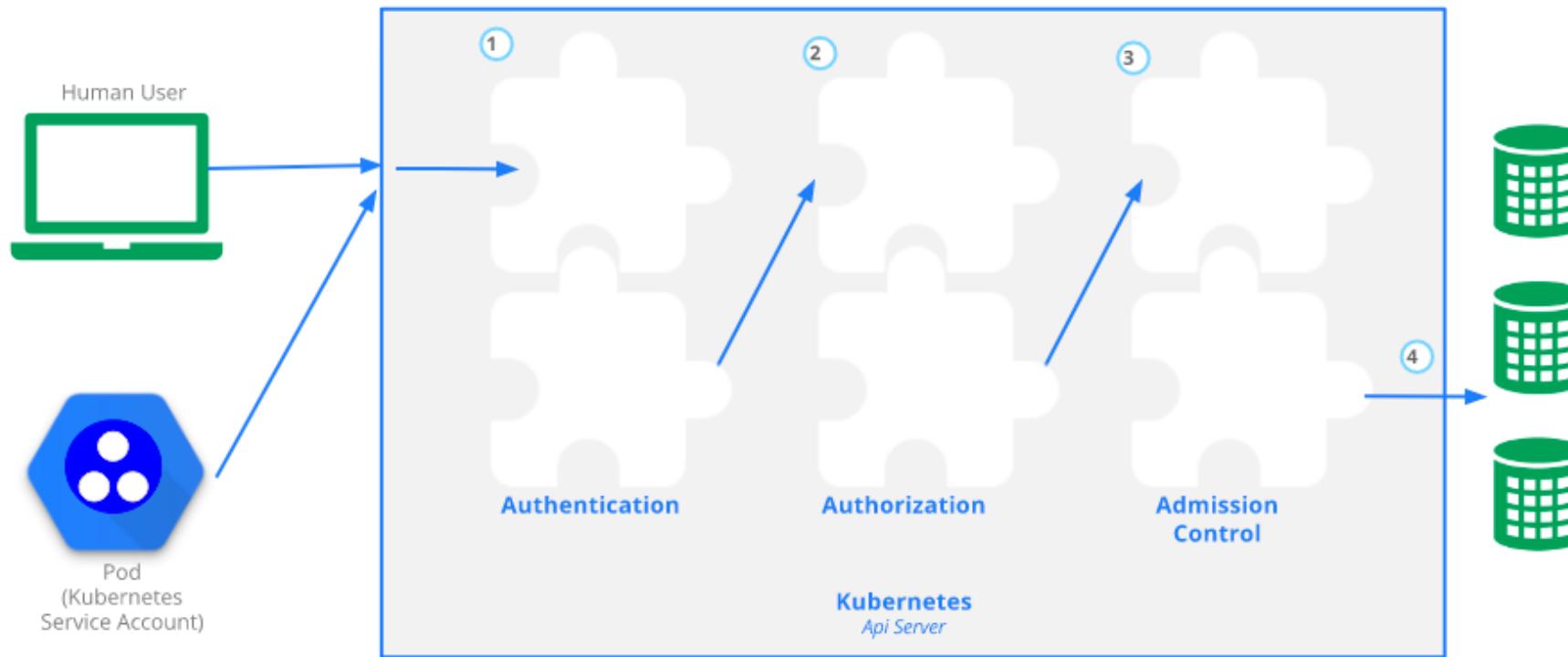
- Etcd v3 的watch机制支持watch某个固定的key，也支持watch一个范围（可以用于模拟目录的结构的watch），所以 watchGroup 包含两种watcher，一种是 key watchers，数据结构是每个key对应一组watcher，另外一种是 range watchers，数据结构是一个 IntervalTree，方便通过区间查找到对应的watcher。
- 同时，每个 WatchableStore 包含两种 watcherGroup，一种是synced，一种是unsynced，前者表示该group的watcher数据都已经同步完毕，在等待新的变更，后者表示该group的watcher数据同步落后于当前最新变更，还在追赶。
- 当 Etcd 收到客户端的watch请求，如果请求携带了revision参数，则比较请求的revision和 store当前的revision，如果大于当前revision，则放入synced组中，否则放入unsynced组。同时 Etcd 会启动一个后台的goroutine持续同步unsynced的watcher，然后将其迁移到synced组。也就是这种机制下，Etcd v3 支持从任意版本开始watch，没有v2的1000条历史event表限制的问题（当然这是指没有compact的情况下）

# API Server

- kube-apiserver是Kubernetes最重要的核心组件之一，主要提供以下的功能
  - 提供集群管理的REST API接口，包括认证授权、数据校验以及集群状态变更等
  - 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）

# 访问控制

- Kubernetes API的每个请求都会经过多阶段的访问控制之后才会被接受，这包括认证、授权以及准入控制（Admission Control）等。



# 认证

- 开启TLS时，所有的请求都需要首先认证。Kubernetes支持多种认证机制，并支持同时开启多个认证插件（只要有一个认证通过即可）。如果认证成功，则用户的username会传入授权模块做进一步授权验证；而对于认证失败的请求则返回HTTP 401。

# 认证插件

- X509证书
  - 使用X509客户端证书只需要API Server启动时配置--client-ca-file=SOMEFILE。在证书认证时，其CN域用作用户名，而组织机构域则用作group名。
- 静态Token文件
  - 使用静态Token文件认证只需要API Server启动时配置--token-auth-file=SOMEFILE。
  - 该文件为csv格式，每行至少包括三列token,username,user id，
    - token,user,uid,"group1,group2,group3"
- 引导Token
  - 引导Token是动态生成的，存储在kube-system namespace的Secret中，用来部署新的Kubernetes集群。
  - 使用引导Token需要API Server启动时配置--experimental-bootstrap-token-auth，并且Controller Manager开启TokenCleaner --controllers=\*,tokencleaner,bootstrapsigner。
  - 在使用kubeadm部署Kubernetes时，kubeadm会自动创建默认token，可通过kubeadm token list命令查询。

# 认证插件

- 静态密码文件
  - 需要API Server启动时配置--basic-auth-file=SOMEFILE，文件格式为csv，每行至少三列password, user, uid，后面是可选的group名，如
    - password,user,uid,"group1,group2,group3"
- Service Account
  - ServiceAccount是Kubernetes自动生成的，并会自动挂载到容器的/run/secrets/kubernetes.io/serviceaccount目录中。
- OpenID
  - OAuth2的认证机制
- OpenStack Keystone密码
  - 需要API Server在启动时指定--experimental-keystone-url=<AuthURL>，而https时还需要设置--experimental-keystone-ca-file=SOMEFILE。
- 匿名请求
  - 如果使用AlwaysAllow以外的认证模式，则匿名请求默认开启，但可用--anonymous-auth=false禁止匿名请求。

# 授权

- 授权主要是用于对集群资源的访问控制，通过检查请求包含的相关属性值，与相对应的访问策略相比较，API请求必须满足某些策略才能被处理。跟认证类似，Kubernetes也支持多种授权机制，并支持同时开启多个授权插件（只要有一个验证通过即可）。如果授权成功，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；对于授权失败的请求则返回HTTP 403。
- Kubernetes授权仅处理以下的请求属性：
  - user, group, extra
  - API、请求方法（如get、post、update、patch和delete）和请求路径（如/api）
  - 请求资源和子资源
  - Namespace
  - API Group
- 目前，Kubernetes支持以下授权插件：
  - ABAC
  - RBAC
  - Webhook
  - Node

# RBAC vs ABAC

- ABAC (Attribute Based Access Control) 本来是不错的概念，但是在 Kubernetes 中的实现比较难于管理和理解，而且需要对 Master 所在节点的 SSH 和文件系统权限，而且要使得对授权的变更成功生效，还需要重新启动 API Server。
- 而 RBAC 的授权策略可以利用 kubectl 或者 Kubernetes API 直接进行配置。RBAC 可以授权给用户，让用户有权进行授权管理，这样就可以无需接触节点，直接进行授权管理。RBAC 在 Kubernetes 中被映射为 API 资源和操作。

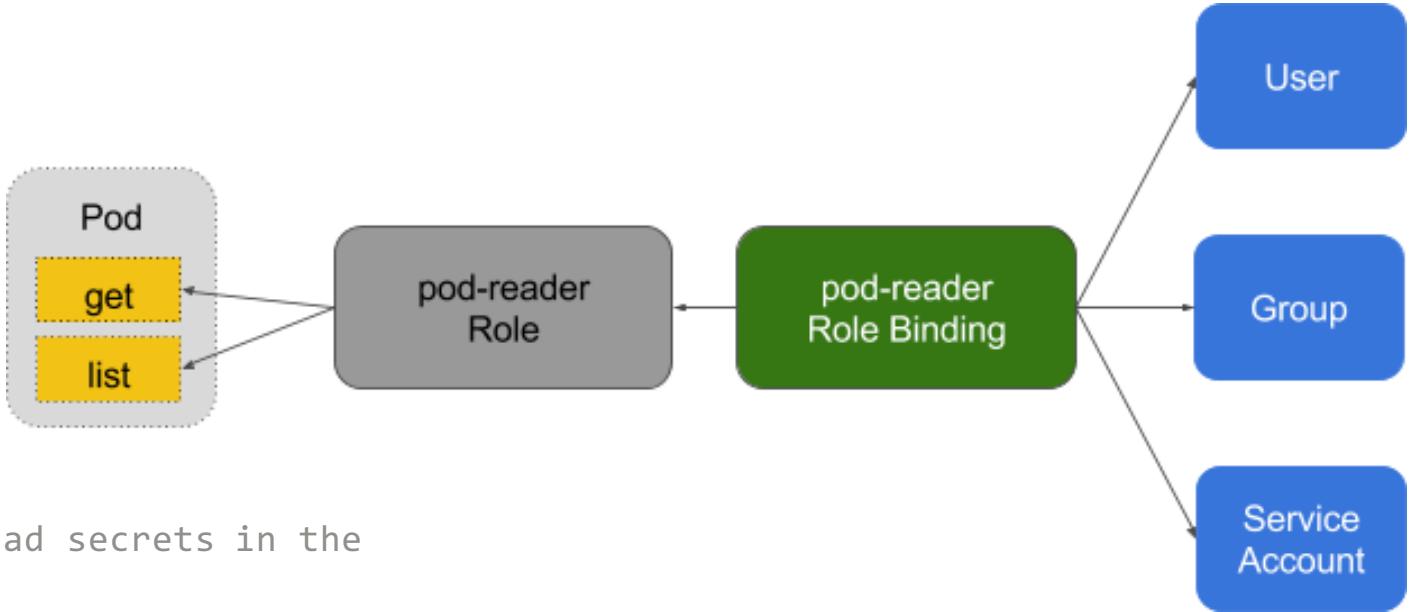
# Role与ClusterRole

- Role（角色）是一系列权限的集合，例如一个角色可以包含读取 Pod 的权限和列出 Pod 的权限。Role只能用来给某个特定namespace中的资源作鉴权，对多namespace和集群级的资源或者是非资源类的API（如/healthz）使用ClusterRole。

```
# Role示例
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

```
# ClusterRole示例
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

# binding



```
# RoleBinding示例 ( 引用ClusterRole )
# This role binding allows "dave" to read secrets in the
"development" namespace.
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-secrets
  namespace: development # This only grants permissions
within the "development" namespace.
subjects:
- kind: User
  name: dave
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

# 准入控制

- 准入控制（Admission Control）在授权后对请求做进一步的验证或添加默认参数。不同于授权和认证只关心请求的用户和操作，准入控制还处理请求的内容，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。
- 准入控制支持同时开启多个插件，它们依次调用，只有全部插件都通过的请求才可以放过进入系统。
-

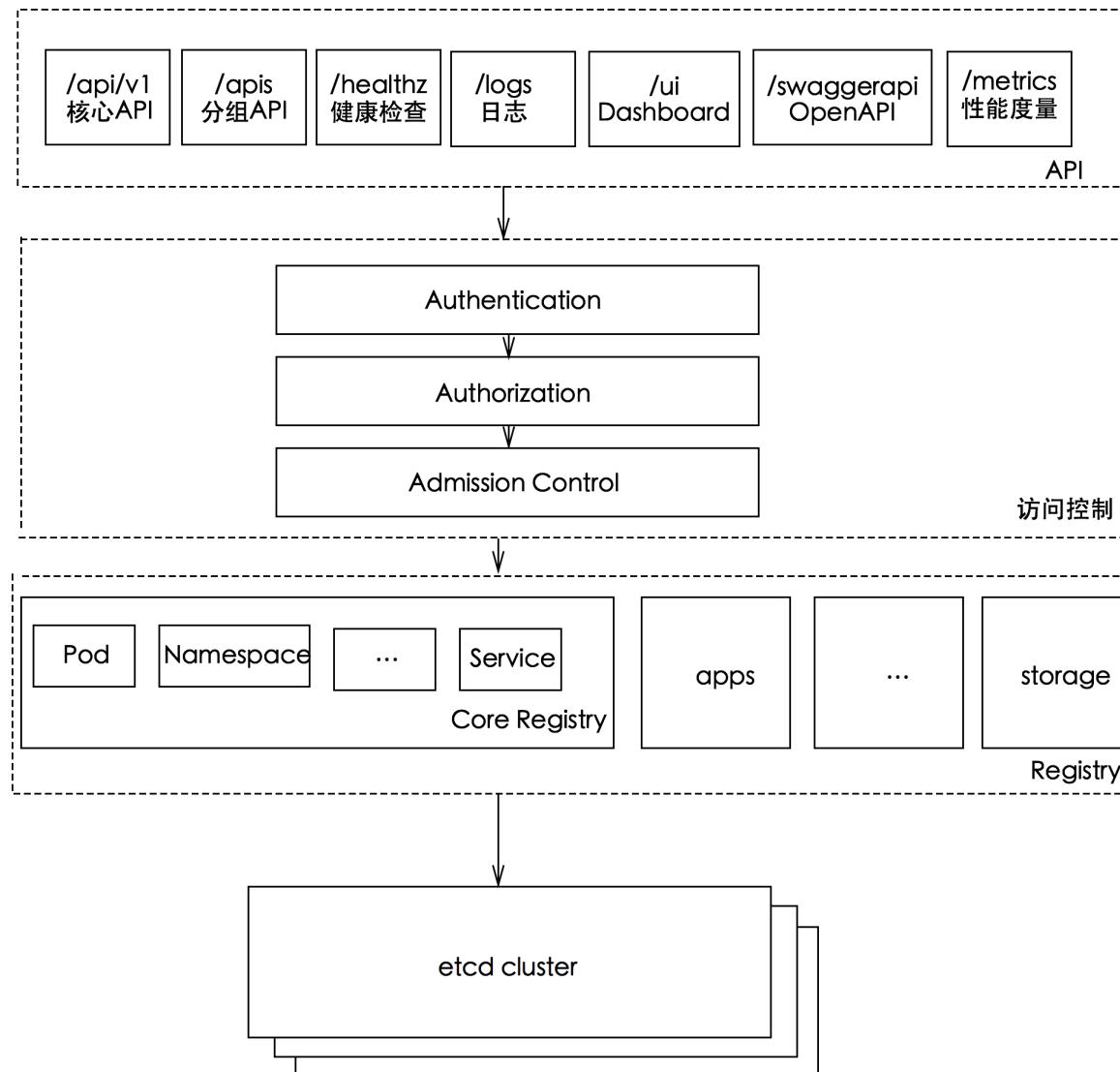
# 准入控制插件

- AlwaysAdmit: 接受所有请求。
- AlwaysPullImages: 总是拉取最新镜像。在多租户场景下非常有用。
- DenyEscalatingExec: 禁止特权容器的exec和attach操作。
- ImagePolicyWebhook: 通过webhook决定image策略, 需要同时配置--admission-control-config-file, 配置文件格式见[这里](#)。
- ServiceAccount: 自动创建默认ServiceAccount, 并确保Pod引用的ServiceAccount已经存在
- SecurityContextDeny: 拒绝包含非法SecurityContext配置的容器
- ResourceQuota: 限制Pod的请求不会超过配额, 需要在namespace中创建一个ResourceQuota对象
- LimitRanger: 为Pod设置默认资源请求和限制, 需要在namespace中创建一个LimitRange对象
- InitialResources: 根据镜像的历史使用记录, 为容器设置默认资源请求和限制
- NamespaceLifecycle: 确保处于termination状态的namespace不再接收新的对象创建请求, 并拒绝请求不存在的namespace
- DefaultStorageClass: 为PVC设置默认StorageClass ([见这里](#))
- DefaultTolerationSeconds: 设置Pod的默认forgiveness tolerance为5分钟
- PodSecurityPolicy: 使用Pod Security Policies时必须开启
- NodeRestriction: 限制kubelet仅可访问node、endpoint、pod、service以及secret、configmap、PV和PVC等相关的资源

# 启动apiserver示例

```
kube-apiserver --feature-gates=AllAlpha=true --runtime-config=api/all=true \
--requestheader-allowed-names=front-proxy-client \
--client-ca-file=/etc/kubernetes/pki/ca.crt \
--allow-privileged=true \
--experimental-bootstrap-token-auth=true \
--storage-backend=etcd3 \
--requestheader-username-headers=X-Remote-User \
--requestheader-extra-headers-prefix=X-Remote-Extra- \
--service-account-key-file=/etc/kubernetes/pki/sa.pub \
--tls-cert-file=/etc/kubernetes/pki/apiserver.crt \
--tls-private-key-file=/etc/kubernetes/pki/apiserver.key \
--kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt \
--requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt \
--insecure-port=8080 \
--admission-
control=NamespaceLifecycle,LimitRanger,ServiceAccount,PersistentVolumeLabel,DefaultStorageClass,
ResourceQuota,DefaultTolerationSeconds \
--requestheader-group-headers=X-Remote-Group \
--kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key \
--secure-port=6443 \
--kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname \
--service-cluster-ip-range=10.96.0.0/12 \
--authorization-mode=RBAC \
--advertise-address=192.168.0.20 --etcd-servers=http://127.0.0.1:2379
```

# kube-apiserver工作原理



# kube-scheduler

- kube-scheduler负责分配调度Pod到集群内的节点上，它监听kube-apiserver，查询还未分配Node的Pod，然后根据调度策略为这些Pod分配节点（更新Pod的NodeName字段）。
- 调度器需要充分考虑诸多的因素：
  - 公平调度
  - 资源高效利用
  - QoS
  - affinity 和 anti-affinity
  - 数据本地化 (data locality)
  - 内部负载干扰 (inter-workload interference)
  - deadlines

# 把pod调度到指定node上

- 可以通过nodeSelector、nodeAffinity、podAffinity以及Taints和tolerations等来将Pod调度到需要的Node上。
- 也可以通过设置nodeName参数，将Pod调度到指定node节点上。
- 比如，使用nodeSelector，首先给Node加上标签：
- kubectl label nodes <your-node-name> disktype=ssd
- 接着，指定该Pod只想运行在带有disktype=ssd标签的Node上：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

# nodeSelector

- 首先给Node打上标签
  - `kubectl label nodes node-01 disktype=ssd`
- 然后在daemonset中指定nodeSelector为disktype=ssd:

```
spec:  
  nodeSelector:  
    disktype: ssd
```

# nodeAffinity

- nodeAffinity目前支持两种：requiredDuringSchedulingIgnoredDuringExecution和preferredDuringSchedulingIgnoredDuringExecution，分别代表必须满足条件和优选条件。
- 比如下面的例子代表调度到包含标签kubernetes.io/e2e-az-name并且值为e2e-az1或e2e-az2的Node上，并且优选还带有标签another-node-label-key=another-node-label-value的Node。

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/e2e-az-name
                operator: In
                values:
                  - e2e-az1
                  - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: another-node-label-key
                operator: In
                values:
                  - another-node-label-value
  containers:
    - name: with-node-affinity
      image: gcr.io/google_containers/pause:2.0
```

# podAffinity

- podAffinity基于Pod的标签来选择Node，仅调度到满足条件Pod所在的Node上，支持podAffinity和podAntiAffinity。这个功能比较绕，以下面的例子为例：
- 如果一个“Node所在Zone中包含至少一个带有security=S1标签且运行中的Pod”，那么可以调度到该Node
- 不调度到“包含至少一个带有security=S2标签且运行中Pod”的Node上

# podAffinity示例

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: security
                operator: In
                values:
                  - S1
      topologyKey: failure-domain.beta.kubernetes.io/zone
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: security
                operator: In
                values:
                  - S2
      topologyKey: kubernetes.io/hostname
  containers:
    - name: with-pod-affinity
      image: gcr.io/google_containers/pause:2.0
```

# Taints和tolerations

- Taints和tolerations用于保证Pod不被调度到不合适的Node上，其中Taint应用于Node上，而toleration则应用于Pod上。
- 目前支持的taint类型
  - NoSchedule：新的Pod不调度到该Node上，不影响正在运行的Pod
  - PreferNoSchedule：soft版的NoSchedule，尽量不调度到该Node上
  - NoExecute：新的Pod不调度到该Node上，并且删除（evict）已在运行的Pod。Pod可以增加一个时间（tolerationSeconds），
- 然而，当Pod的Tolerations匹配Node的所有Taints的时候可以调度到该Node上；当Pod是已经运行的时候，也不会被删除（evicted）。另外对于NoExecute，如果Pod增加了一个tolerationSeconds，则会在该时间之后才删除Pod。

# 优先级调度

- 从v1.8开始，kube-scheduler支持定义Pod的优先级，从而保证高优先级的Pod优先调度。开启方法为
  - apiserver配置--feature-gates=PodPriority=true 和 --runtime-config=scheduling.k8s.io/v1alpha1=true
  - kube-scheduler配置--feature-gates=PodPriority=true

# PriorityClass

- 在指定Pod的优先级之前需要先定义一个PriorityClass（非namespace资源），如

```
apiVersion: v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service pods only."
```

# 为pod设置priority

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

# 多调度器

- 如果默认的调度器不满足要求，还可以部署自定义的调度器。并且，在整个集群中还可以同时运行多个调度器实例，通过podSpec.schedulerName来选择使用哪一个调度器（默认使用内置的调度器）。
-

# 调度器

- kube-scheduler调度分为两个阶段，predicate和priority
  - predicate：过滤不符合条件的节点
  - priority：优先级排序，选择优先级最高的节点

# predicates策略

- PodFitsPorts: 同PodFitsHostPorts
- PodFitsHostPorts: 检查是否有Host Ports冲突
- PodFitsResources: 检查Node的资源是否充足，包括允许的Pod数量、CPU、内存、GPU个数以及其他OpaqueIntResources
- HostName: 检查pod.Spec.NodeName是否与候选节点一致
- MatchNodeSelector: 检查候选节点的pod.Spec.NodeSelector是否匹配
- NoVolumeZoneConflict: 检查volume zone是否冲突
- MaxEBSVolumeCount: 检查AWS EBS Volume数量是否过多（默认不超过39）
- MaxGCEPDVolumeCount: 检查GCE PD Volume数量是否过多（默认不超过16）
- MaxAzureDiskVolumeCount: 检查Azure Disk Volume数量是否过多（默认不超过16）
- MatchInterPodAffinity: 检查是否匹配Pod的亲和性要求
- NoDiskConflict: 检查是否存在Volume冲突，仅限于GCE PD、AWS EBS、Ceph RBD以及iSCSI
- GeneralPredicates: 分为noncriticalPredicates和EssentialPredicates。noncriticalPredicates中包含PodFitsResources，EssentialPredicates中包含PodFitsHost, PodFitsHostPorts和PodSelectorMatches。
- PodToleratesNodeTaints: 检查Pod是否容忍Node Taints
- CheckNodeMemoryPressure: 检查Pod是否可以调度到MemoryPressure的节点上
- CheckNodeDiskPressure: 检查Pod是否可以调度到DiskPressure的节点上
- NoVolumeNodeConflict: 检查节点是否满足Pod所引用的Volume的条件

# priorities策略

- SelectorSpreadPriority: 优先减少节点上属于同一个Service或Replication Controller的Pod数量
- InterPodAffinityPriority: 优先将Pod调度到相同的拓扑上（如同一个节点、Rack、Zone等）
- LeastRequestedPriority: 优先调度到请求资源少的节点上
- BalancedResourceAllocation: 优先平衡各节点的资源使用
- NodePreferAvoidPodsPriority: alpha.kubernetes.io/preferAvoidPods字段判断,权重为10000，避免其他优先级策略的影响
- NodeAffinityPriority: 优先调度到匹配NodeAffinity的节点上
- TaintTolerationPriority: 优先调度到匹配TaintToleration的节点上
- ServiceSpreadingPriority: 尽量将同一个service的Pod分布到不同节点上，已经被SelectorSpreadPriority替代[默认未使用]
- EqualPriority: 将所有节点的优先级设置为1[默认未使用]
- ImageLocalityPriority: 尽量将使用大镜像的容器调度到已经下拉了该镜像的节点上[默认未使用]
- MostRequestedPriority: 尽量调度到已经使用过的Node上，特别适用于cluster-autoscaler[默认未使用]

# Controller Manager

- Replication Controller
- Node Controller
- CronJob Controller
- Daemon Controller
- Deployment Controller
- Endpoint Controller
- Garbage Collector
- Namespace Controller
- Job Controller
- Pod AutoScaler
- RelicaSet
- Service Controller
- ServiceAccount Controller
- StatefulSet Controller
- Volume Controller
- Resource quota Controller

# Kubelet

- 每个节点上都运行一个kubelet服务进程， 默认监听10250端口， 接收并执行master发来的指令， 管理Pod及Pod中的容器。每个kubelet进程会在API Server上注册节点自身信息， 定期向master节点汇报节点的资源使用情况，并通过cAdvisor监控节点和容器的资源。
-

# 节点管理

- 节点管理主要是节点自注册和节点状态更新：
  - Kubelet可以通过设置启动参数 `--register-node` 来确定是否向API Server注册自己；
  - 如果Kubelet没有选择自注册模式，则需要用户自己配置Node资源信息，同时需要告知Kubelet集群上的API Server的位置；
  - Kubelet在启动时通过API Server注册节点信息，并定时向API Server发送节点新消息，API Server在接收到新消息后，将信息写入etcd
-

# Pod管理

- 获取Pod清单
  - 文件：启动参数 `--config` 指定的配置目录下的文件(默认`/etc/kubernetes/manifests/`)。该文件每20秒重新检查一次（可配置）。
  - HTTP endpoint (URL)：启动参数 `--manifest-url` 设置。每20秒检查一次这个端点（可配置）。
  - API Server：通过API Server监听etcd目录，同步Pod清单。
  - HTTP server：kubelet侦听HTTP请求，并响应简单的API以提交新的Pod清单。

# 通过API Server获取Pod清单及创建Pod的过程

- Kubelet通过API Server Client(Kubelet启动时创建)使用Watch加List的方式监听"/registry/nodes/\$当前节点名"和"/registry/pods"目录，将获取的信息同步到本地缓存中。
- Kubelet监听etcd，所有针对Pod的操作都将会被Kubelet监听到。如果发现有新的绑定到本节点的Pod，则按照Pod清单的要求创建该Pod。
- 如果发现本地的Pod被修改，则Kubelet会做出相应的修改，比如删除Pod中某个容器时，则通过Docker Client删除该容器。如果发现删除本节点的Pod，则删除相应的Pod，并通过Docker Client删除Pod中的容器。

# Kubelet读取监听到的信息，如果是创建和修改Pod任务，则执行如下处理：

- 为该Pod创建一个数据目录；
- 从API Server读取该Pod清单；
- 为该Pod挂载外部卷；
- 下载Pod用到的Secret；
- 检查已经在节点上运行的Pod，如果该Pod没有容器或Pause容器没有启动，则先停止Pod里所有容器的进程。如果在Pod中有需要删除的容器，则删除这些容器；
- 用“kubernetes/pause”镜像为每个Pod创建一个容器。Pause容器用于接管Pod中所有其他容器的网络。每创建一个新的Pod，Kubelet都会先创建一个Pause容器，然后创建其他容器。
- 为Pod中的每个容器做如下处理：
  - 为容器计算一个hash值，然后用容器的名字去Docker查询对应容器的hash值。若查找到容器，且两者hash值不同，则停止Docker中容器的进程，并停止与之关联的Pause容器的进程；若两者相同，则不做任何处理；
  - 如果容器被终止了，且容器没有指定的restartPolicy，则不做任何处理；
  - 调用Docker Client下载容器镜像，调用Docker Client运行容器。

# Static Pod

- 所有以非API Server方式创建的Pod都叫Static Pod。Kubelet将Static Pod的状态汇报给API Server， API Server为该Static Pod创建一个Mirror Pod和其相匹配。Mirror Pod的状态将真实反映Static Pod的状态。当Static Pod被删除时，与之相对应的Mirror Pod也会被删除。
-

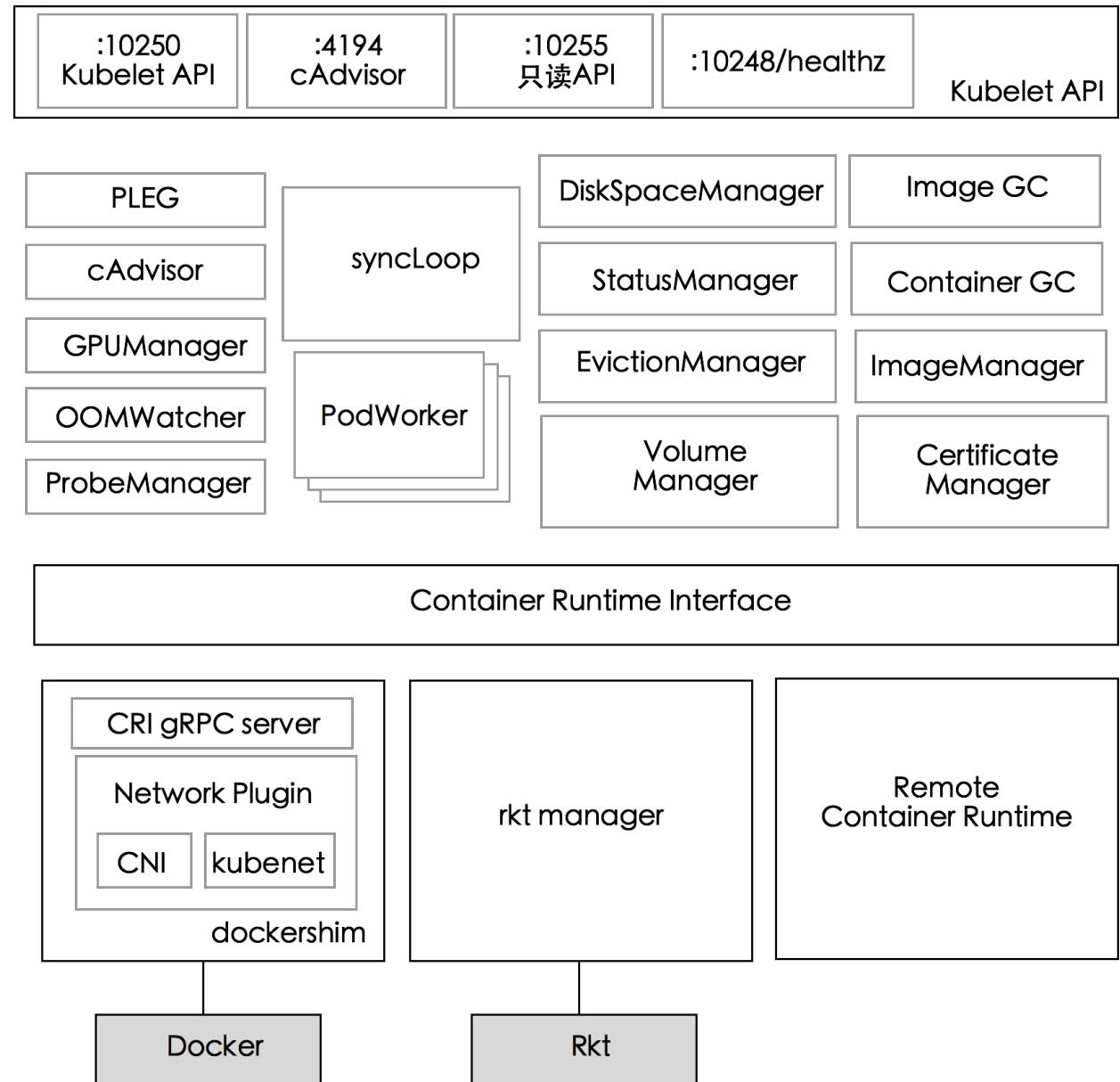
# 容器健康检查

- Pod通过两类探针检查容器的健康状态：
  - (1) LivenessProbe 探针：用于判断容器是否健康，告诉Kubelet一个容器什么时候处于不健康的状态。如果LivenessProbe探针探测到容器不健康，则Kubelet将删除该容器，并根据容器的重启策略做相应的处理。如果一个容器不包含LivenessProbe探针，那么Kubelet认为该容器的LivenessProbe探针返回的值永远是“Success”；
  - (2)ReadinessProbe：用于判断容器是否启动完成且准备接收请求。如果ReadinessProbe探针探测到失败，则Pod的状态将被修改。Endpoint Controller将从Service的Endpoint中删除包含该容器所在Pod的IP地址的Endpoint条目。
- Kubelet定期调用容器中的LivenessProbe探针来诊断容器的健康状况。LivenessProbe包含如下三种实现方式：
  - ExecAction：在容器内部执行一个命令，如果该命令的退出状态码为0，则表明容器健康；
  - TCPSocketAction：通过容器的IP地址和端口号执行TCP检查，如果端口能被访问，则表明容器健康；
  - HTTPGetAction：通过容器的IP地址和端口号及路径调用HTTP GET方法，如果响应的状态码大于等于200且小于400，则认为容器状态健康。
- LivenessProbe探针包含在Pod定义的spec.containers.{某个容器}中。

# cAdvisor资源监控

- Kubernetes集群中，应用程序的执行情况可以在不同的级别上监测到，这些级别包括：容器、Pod、Service和整个集群。
- Heapster项目为Kubernetes提供了一个基本的监控平台，它是集群级别的监控和事件数据集成器(Aggregator)。Heapster以Pod的方式运行在集群中，Heapster通过Kubelet发现所有运行在集群中的节点，并查看来自这些节点的资源使用情况。Kubelet通过cAdvisor获取其所在节点及容器的数据。Heapster通过带着关联标签的Pod分组这些信息，这些数据将被推到一个可配置的后端，用于存储和可视化展示。支持的后端包括InfluxDB(使用Grafana实现可视化)和Google Cloud Monitoring。
- cAdvisor是一个开源的分析容器资源使用率和性能特性的代理工具，已集成到Kubernetes代码中。cAdvisor自动查找所有在其所在节点上的容器，自动采集CPU、内存、文件系统和网络使用的统计信息。cAdvisor通过它所在节点机的Root容器，采集并分析该节点机的全面使用情况。  
cAdvisor通过其所在节点机的4194端口暴露一个简单的UI。

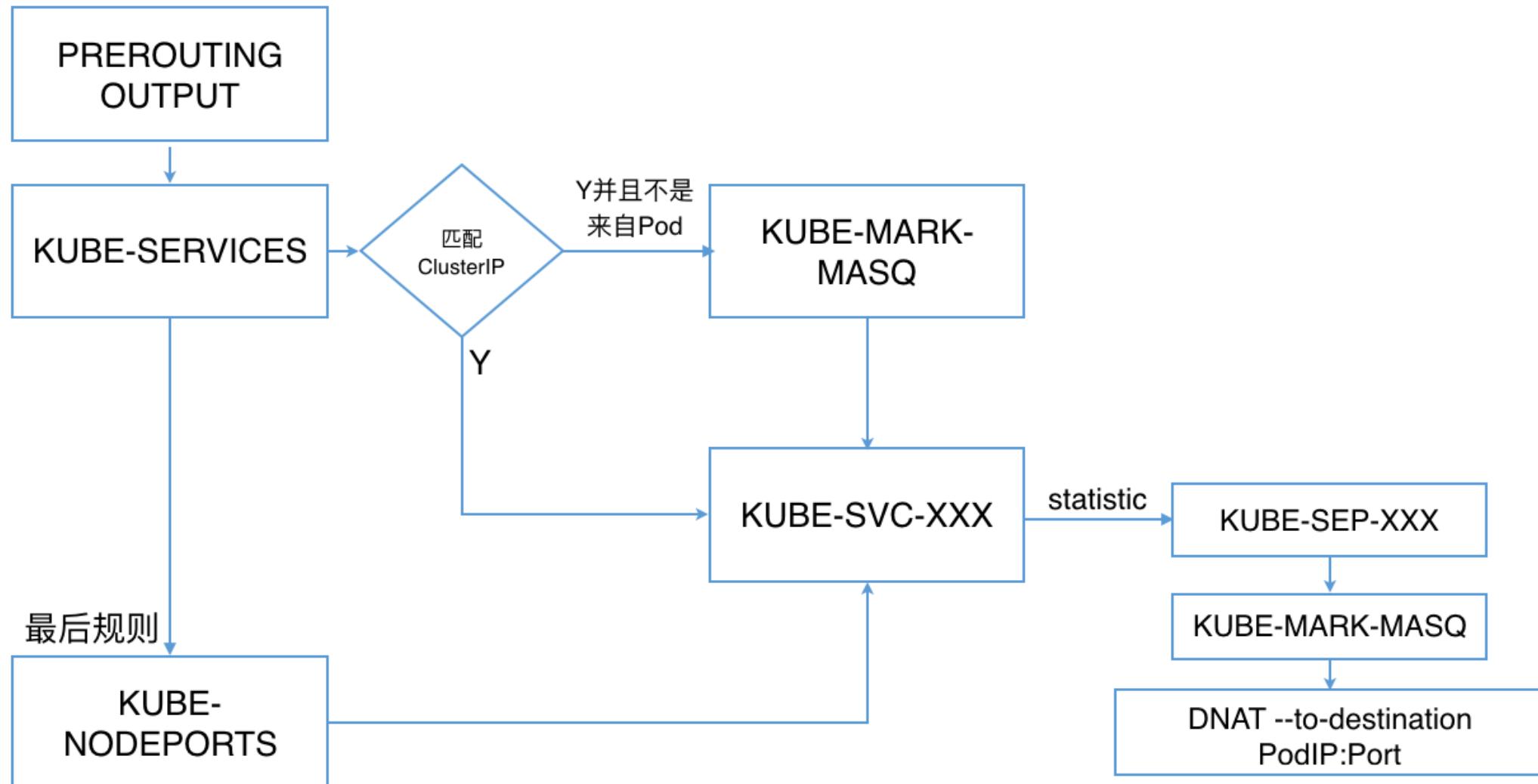
# Kubelet组件



# kube-proxy

- 每台机器上都运行一个kube-proxy服务，它监听API server中service和endpoint的变化情况，并通过iptables等来为服务配置负载均衡（仅支持TCP和UDP）。
- kube-proxy可以直接运行在物理机上，也可以以static pod或者daemonset的方式运行。
- kube-proxy当前支持一下几种实现
  - userspace：最早的负载均衡方案，它在用户空间监听一个端口，所有服务通过iptables转发到这个端口，然后在其内部负载均衡到实际的Pod。该方式最主要的问题是效率低，有明显的性能瓶颈。
  - iptables：目前推荐的方案，完全以iptables规则的方式来实现service负载均衡。该方式最主要的问题是在服务多的时候产生太多的iptables规则，非增量式更新会引入一定的时延，大规模情况下有明显的性能问题
  - ipvs：为解决iptables模式的性能问题，v1.8新增了ipvs模式，采用增量式更新，并可以保证service更新期间连接保持不断开
  - winuserspace：同userspace，但仅工作在windows上

# Kubernetes Iptables规则



# Iptables示例

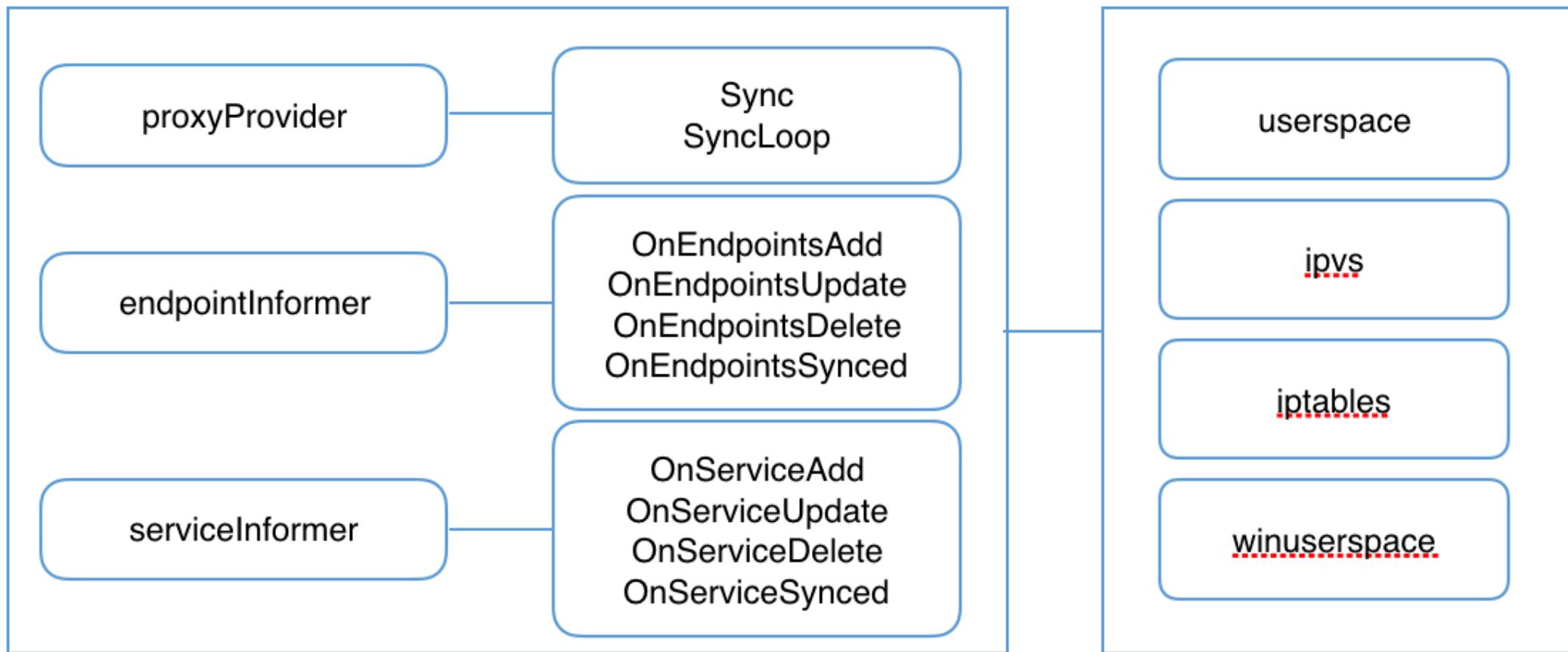
```
-A KUBE-MARK-DROP -j MARK --set-xmark 0x8000/0x8000
-A KUBE-MARK-MASQ -j MARK --set-xmark 0x4000/0x4000
-A KUBE-POSTROUTING -m comment --comment "kubernetes service traffic requiring SNAT" -m mark --mark 0x4000/0x4000 -j MASQUERADE

-A KUBE-SEP-55QZ6T7MF3AHPOOB -s 10.244.1.6/32 -m comment --comment "default/http:" -j KUBE-MARK-MASQ
-A KUBE-SEP-55QZ6T7MF3AHPOOB -p tcp -m comment --comment "default/http:" -m tcp -j DNAT --to-destination 10.244.1.6:80

-A KUBE-SEP-KJZJRL2KRWMXNR3J -s 10.244.1.5/32 -m comment --comment "default/http:" -j KUBE-MARK-MASQ
-A KUBE-SEP-KJZJRL2KRWMXNR3J -p tcp -m comment --comment "default/http:" -m tcp -j DNAT --to-destination 10.244.1.5:80

-A KUBE-SERVICES -d 10.101.85.234/32 -p tcp -m comment --comment "default/http: cluster IP" -m tcp --dport 80 -j KUBE-SVC-7IMAZDGB2ONQNK4Z
-A KUBE-SVC-7IMAZDGB2ONQNK4Z -m comment --comment "default/http:" -m statistic --mode random --probability 0.500000000000 -j KUBE-SEP-KJZJRL2KRWMXNR3J
-A KUBE-SVC-7IMAZDGB2ONQNK4Z -m comment --comment "default/http:" -j KUBE-SEP-55QZ6T7MF3AHPOOB
```

# kube-proxy工作原理

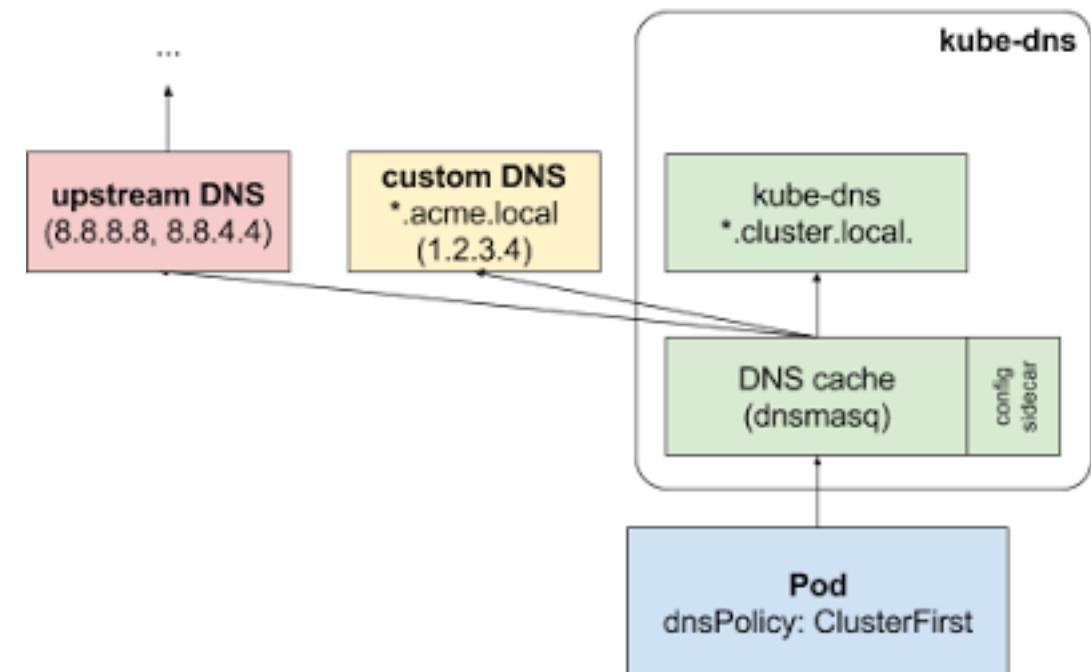


# kube-dns

- 支持的DNS格式
- Service
  - A record: 生成my-svc.my-namespace.svc.cluster.local，解析IP分为两种情况
    - 普通Service解析为Cluster IP
    - Headless Service解析为指定的Pod IP列表
  - SRV record: 生成\_my-port-name.\_my-port-protocol.my-svc.my-namespace.svc.cluster.local
- Pod
  - A record: pod-ip-address.my-namespace.pod.cluster.local
  - 指定hostname和subdomain: hostname.custom-subdomain.default.svc.cluster.local

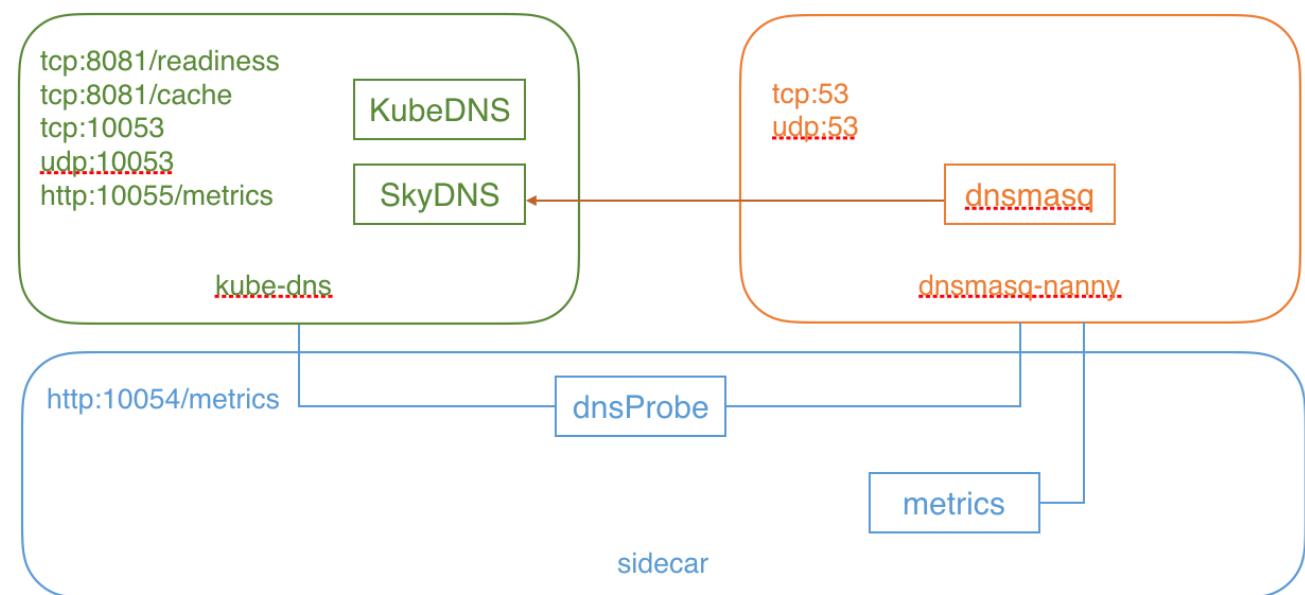
# 私有DNS服务器和上游DNS服务器

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {"acme.local": ["1.2.3.4"]}
  upstreamNameservers: |
    ["8.8.8.8", "8.8.4.4"]
```



# kube-dns工作原理

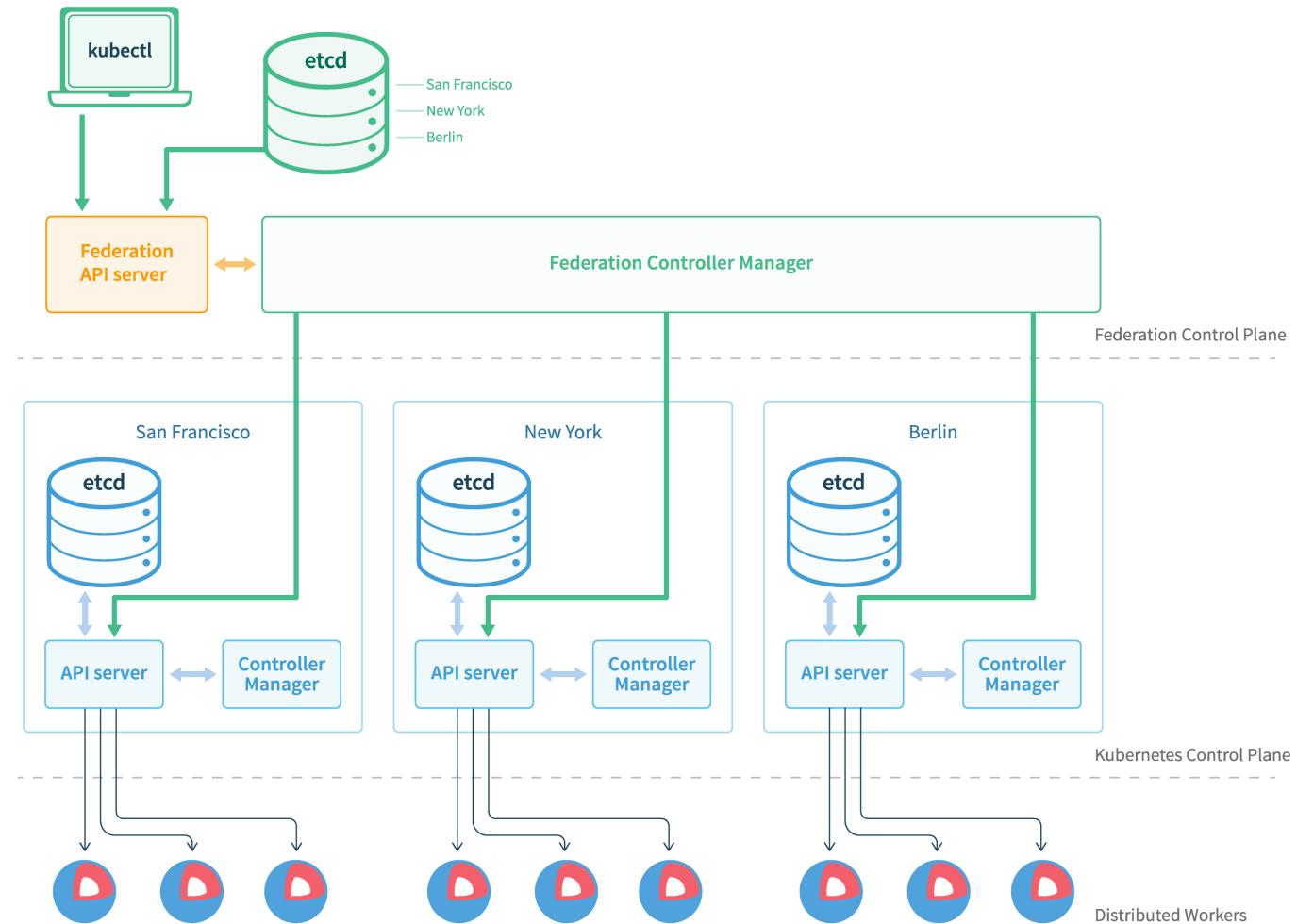
- kube-dns由三个容器构成：
- kube-dns：DNS服务的核心组件，主要由 KubeDNS和SkyDNS组成
  - KubeDNS负责监听Service和Endpoint的变化情况，并将相关的信息更新到SkyDNS中
  - SkyDNS负责DNS解析，监听在10053端口(tcp/udp)，同时也监听在10055端口提供metrics
  - kube-dns还监听了8081端口，以供健康检查使用
- dnsmasq-nanny：负责启动dnsmasq，并在配置发生变化时重启dnsmasq
  - dnsmasq的upstream为SkyDNS，即集群内部的 DNS解析由SkyDNS负责
- sidecar：负责健康检查和提供DNS metrics（监听在10054端口）



# Federation

- 在云计算环境中，服务的作用距离范围从近到远一般可以有：同主机（Host, Node）、跨主机同可用区（Available Zone）、跨可用区同地区（Region）、跨地区同服务商（Cloud Service Provider）、跨云平台。
- K8s的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足K8s的调度和计算存储连接要求。而集群联邦（Federation）就是为提供跨Region跨服务商K8s集群服务而设计的。
- 每个Federation有自己的分布式存储、API Server和Controller Manager。用户可以通过Federation的API Server注册该Federation的成员K8s Cluster。当用户通过Federation的API Server创建、更改API对象时，Federation API Server会在自己所有注册的子K8s Cluster都创建一份对应的API对象。在提供业务请求服务时，K8s Federation会先在自己的各个子Cluster之间做负载均衡，而对于发送到某个具体K8s Cluster的业务请求，会依照这个K8s Cluster独立提供服务时一样的调度模式去做K8s Cluster内部的负载均衡。而Cluster之间的负载均衡是通过域名服务的负载均衡来实现的。

# Federation 架构



# 注册集群和查询

- kubernetes集群可以通过kubefed join命令加入集群联邦：
  - \$ kubefed join federation – cluster-context=myclustercontext
- 查询注册到Federation的kubernetes集群列表
  - \$ kubectl --context=federation get clusters

# 集群联邦使用

- Federated ConfigMap
- Federated Service
- Federated DaemonSet
- Federated Deployment
- Federated Ingress
- Federated Namespaces
- Federated ReplicaSets
- Federated Secrets
- Federated Events (仅存在federation控制平面)
- Federated Jobs (v1.8+)
- Federated Horizontal Pod Autoscaling (HPA, v1.8+)

# ClusterSelector

- federation.alpha.kubernetes.io/cluster-selector为新对象选择kubernetes集群

```
metadata:  
annotations:  
  federation.alpha.kubernetes.io/cluster-selector: '[{"key": "pci", "operator":  
    "In", "values": ["true"]}, {"key": "environment", "operator": "NotIn", "values":  
    ["test"]}]'
```

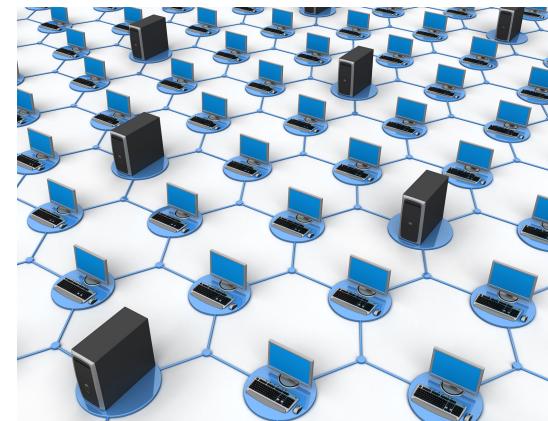
# 高可用(99.999)

- 永远记住，硬件是不可靠的
- 按照经验，最容易发生故障的点是：

存储



网络



# 高可用

- 高可用，英文原文为High Availability，简称HA，简单的说，集群（cluster）就是一组计算机，它们作为一个整体向用户提供一组网络资源。这些单个的计算机系统就是集群的节点（node）。高可用性集群（HA cluster）是指如单系统一样地运行并支持（计算机）持续正常运行的一个主机群。
- 高可用集群的出现是为了使集群的整体服务尽可能可用，从而减少由计算机硬件和软件易错性所带来的损失。如果某个节点失效，它的备援节点将在几秒钟的时间内接管它的职责。因此，对于用户而言，集群永远不会停机。高可用集群软件的主要作用就是实现故障检查和业务切换的自动化。
- 可用性（availability）是关于系统可供使用时间的表述，以不可用时间为衡量指标。不可用时间越短，可用性越高。通常用n个9来描述。比如4个9的可用性（99.99%），是指一年中不可用时间在52分钟内，平均每周不可用时间在1分钟。

# 高可用

Level of Availability	Percent of Uptime	Downtime per Year	Downtime per Day
1 Nine	90%	36.5 days	2.4 hrs.
2 Nines	99%	3.65 days	14 min.
3 Nines	99.9%	8.76 hrs.	86 sec.
4 Nines	99.99%	52.6 min.	8.6 sec.
5 Nines	99.999%	5.25 min.	.86 sec.
6 Nines	99.9999%	31.5 sec.	8.6 msec

- 要谈可用性，首先必须承认所有东西都有不可用的时候，只是不可用程度而已。一般来说，我们的观念里一个服务至少要做到99.9% 才称为基本上可用，是合格性产品。否则基本很难被别人使用。
- 从3个9迈向4个9，从8小时一下缩短到52.6分钟的不可用时间，是一个很大的进步。

# 高可用

- 高可用是一个系统性问题，需要从上到下的系统化解决
  - 供电
  - 存储
  - 网络
  - 计算节点
  - 服务容错
  - 部署
  - 负载均衡
  - 自我修复
  - 发布流程
  - 变更控制
  - 运维成熟度
  - 流量切换
  - .....

# 高可用

- MTBF: Mean time between Failures。用通俗的话讲，就是一个东西有多不可靠，多长时间坏一次。
- MTTR: Mean time to recover。意思就是一旦坏了，恢复服务的时间需要多长。

$$Availability = f(MTBF, MTTR)$$

- 一个服务的可用度，取决于MTBF和MTTR这两个因子。从这个公式出发，结合实际情况，就很好理清高可用架构的基本路数了。那就是：要么提高MTBF，要么降低 MTTR。

# 高可用性方案

- 提高冗余度，多实例运行，用资源换可用性
- $N + 2$ 
  - $N + 2$ 就是说平时如果一个服务需要1个实例正常提供服务，那么我们就在生产环境上应该部署 $1 + 2 = 3$ 个节点。大家可能觉得 $N + 1$ 很合理，也就是有个热备份系统，比较能够接受。但是你要想到，服务 $N + 1$ 部署只能提供热备容灾，发布的时候就失去保护了。
  - 从另一个角度来讲，服务 $N + 2$ 说的是在丢失两个最大的实例的同时，依然可以维持业务的正常运转。
  - 这其实就是最近常说的两地三中心的概念有点像。
- 实例之间对等、独立
  - 一大一小，或者相互依赖都不是真的 $N + 2$ 。如果两地三中心的一个中心是需要几小时才能迁移过去的，那他就不是一个高可用性部署，顶多算异地灾备。

# 高可用性方案

- 流量控制
  - 想做到高可用，必须拥有一套非常可靠的流量控制系统。这套系统按常见的维度，最好能按业务维度来调度流量。
  - 静态流控：主要针对客户端访问速率进行控制，它通常根据服务质量等级协定（SLA）中约定的QPS做全局流量控制。
  - 动态流控：它的最终目标是为了保命，并不是对流量或者访问速度做精确控制。当系统负载压力非常大时，系统进入过负载状态，可能是CPU、内存资源已经过载，也可能是应用进程内部的资源几乎耗尽，如果继续全量处理业务，可能会导致长时间的Full GC、消息严重积压或者应用进程宕机，最终将压力转移到集群其它节点，引起级联故障。触发动态流控的因子是资源，资源又分为系统资源和应用资源两大类，根据不同的资源负载情况，动态流控又分为多个级别，每个级别流控系数都不同，也就是被拒绝掉的消息比例不同。每个级别都有相应的流控阈值，这个阈值通常支持在线动态调整。

# 高可用性方案

- 线下测试 (Offline Test)
  - 线下测试永远比线上调试容易一百倍，也安全一百倍。
  - 可用性的阶段性提高，不是靠运维团队，而是靠产品团队。能在线下完成的测试，绝不到线上去实验。
- 灰度发布
  - 灰度发布是速度与安全性作为妥协。他是发布众多保险的最后一道，而不是唯一的一道。如果只是为了灰度而灰度，故意人为的拖慢进度，反而造成线上多版本长期间共存，有可能会引入新的问题。
  - 测试代码要充足。
  - 监控体系要成熟。
  - 平台要支持。
- 服务必须对回滚提供支持
  - 这点不允许商量！
- 不允许没有监控的系统上线！

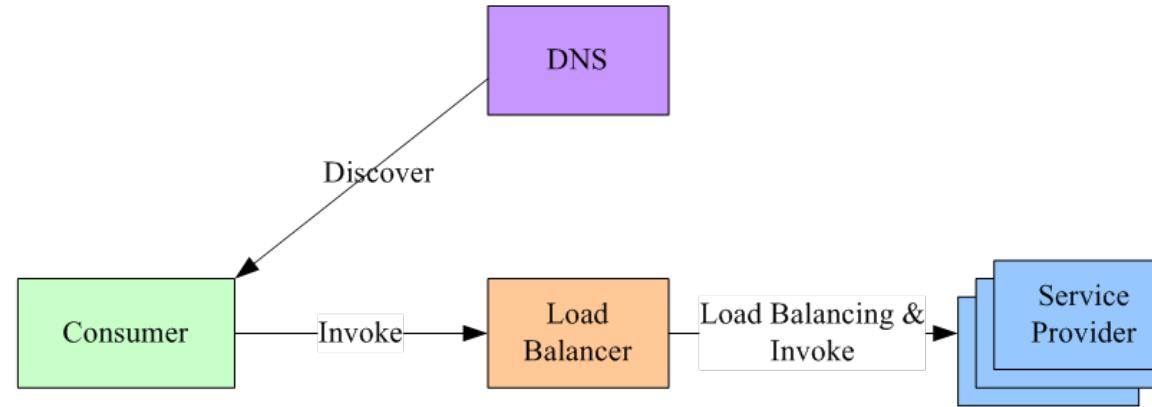
# Kubernetes HA

- Kubernetes本身的HA
  - etcd
  - apiserver
  - scheduler
  - controller-manager
- Kubernetes提供的HA
  - pod probe
  - replication controller
  - service
  - federation

# 服务发现

- 微服务架构是由一系列职责单一的细粒度服务构成的分布式网状结构，服务之间通过轻量机制进行通信，这时候必然引入一个服务注册发现问题，也就是说服务提供方要注册通告服务地址，服务的调用方要能发现目标服务。
- 同时服务提供方一般以集群方式提供服务，也就引入了负载均衡和健康检查问题。

# 集中式LB服务发现

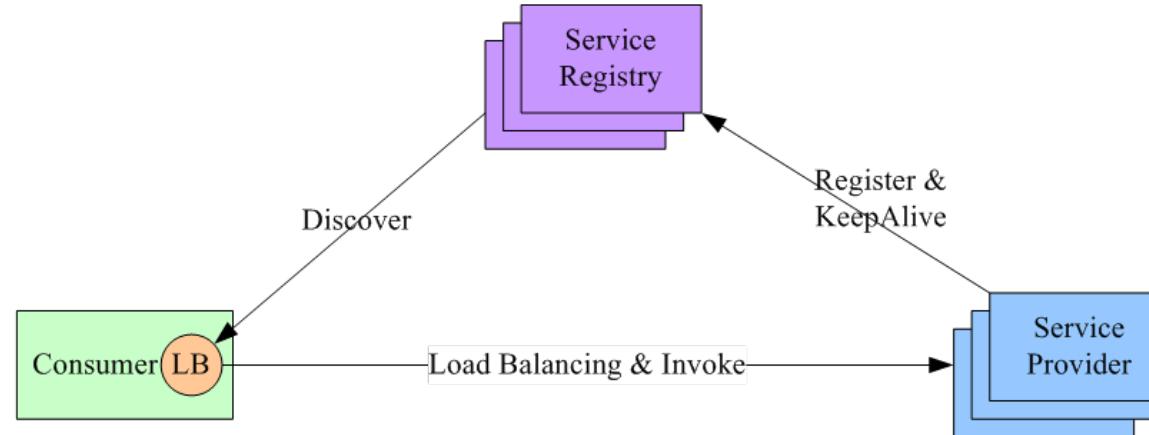


- 在服务消费者和服务提供者之间有一个独立的LB。
- LB上有所有服务的地址映射表，通常由运维配置注册。
- 当服务消费方调用某个目标服务时，它向LB发起请求，由LB以某种策略（比如Round-Robin）做负载均衡后将请求转发到目标服务。
- LB一般具备健康检查能力，能自动摘除不健康的服务实例。
- 服务消费方通过DNS发现LB，运维人员为服务配置一个DNS域名，这个域名指向LB。

# 集中式LB服务发现

- 集中式LB方案实现简单，在LB上也容易做集中式的访问控制，这一方案目前还是业界主流。
- 集中式LB的主要问题是单点问题，所有服务调用流量都经过LB，当服务数量和调用量大的时候，LB容易成为瓶颈，且一旦LB发生故障对整个系统的影响是灾难性的。
- LB在服务消费方和服务提供方之间增加了一跳(hop)，有一定性能开销。

# 进程内LB服务发现

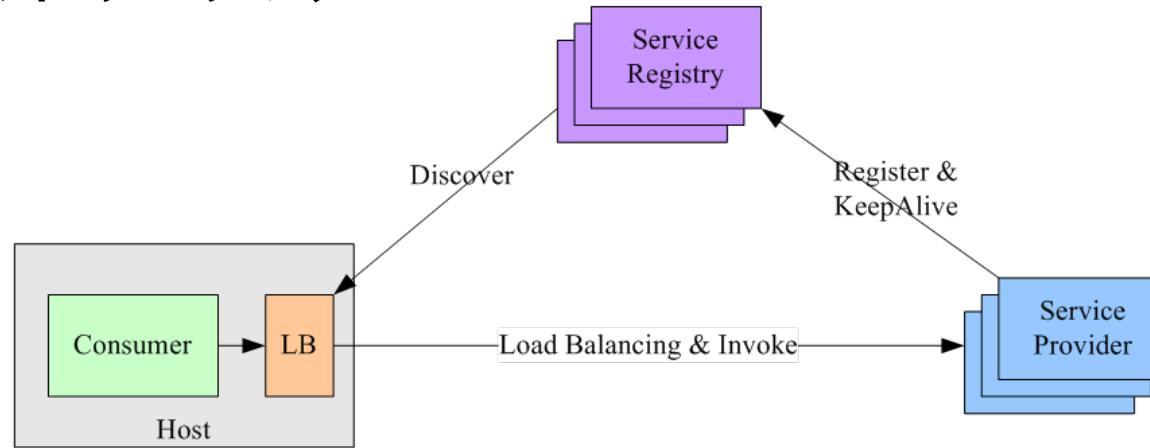


- 进程内LB方案将LB的功能以库的形式集成到服务消费方进程里头，该方案也被称为客户端负载方案。
- 服务注册表(Service Registry)配合支持服务自注册和自发现，服务提供方启动时，首先将服务地址注册到服务注册表（同时定期报心跳到服务注册表以表明服务的存活状态）。
- 服务消费方要访问某个服务时，它通过内置的LB组件向服务注册表查询（同时缓存并定期刷新）目标服务地址列表，然后以某种负载均衡策略选择一个目标服务地址，最后向目标服务发起请求。
- 这一方案对服务注册表的可用性(Availability)要求很高，一般采用能满足高可用分布式一致的组件（例如Zookeeper, Consul, Etcd等）来实现。

# 进程内LB服务发现

- 进程内LB是一种分布式模式，LB和服务发现能力被分散到每一个服务消费者的进程内部，同时服务消费方和服务提供方之间是直接调用，没有额外开销，性能比较好。该方案以客户库(Client Library)的方式集成到服务调用方进程里头，如果企业内有多种不同的语言栈，就要配合开发多种不同的客户端，有一定的研发和维护成本。
- 一旦客户端跟随服务调用方发布到生产环境中，后续如果要对客户库进行升级，势必要求服务调用方修改代码并重新发布，所以该方案的升级推广有不小的阻力。

# 独立LB进程服务发现



- 针对进程内LB模式的不足而提出的一种折中方案，原理和第二种方案基本类似。
- 不同之处是，将LB和服务发现功能从进程内移出来，变成主机上的一个独立进程，主机上的一个或者多个服务要访问目标服务时，他们都通过同一主机上的独立LB进程做服务发现和负载均衡。
- LB独立进程可以进一步与服务消费方进行解耦，以独立集群的形式提供高可用的负载均衡服务。
- 这种模式可以称之为真正的“软负载（Soft Load Balancing）”。

# 独立LB进程服务发现

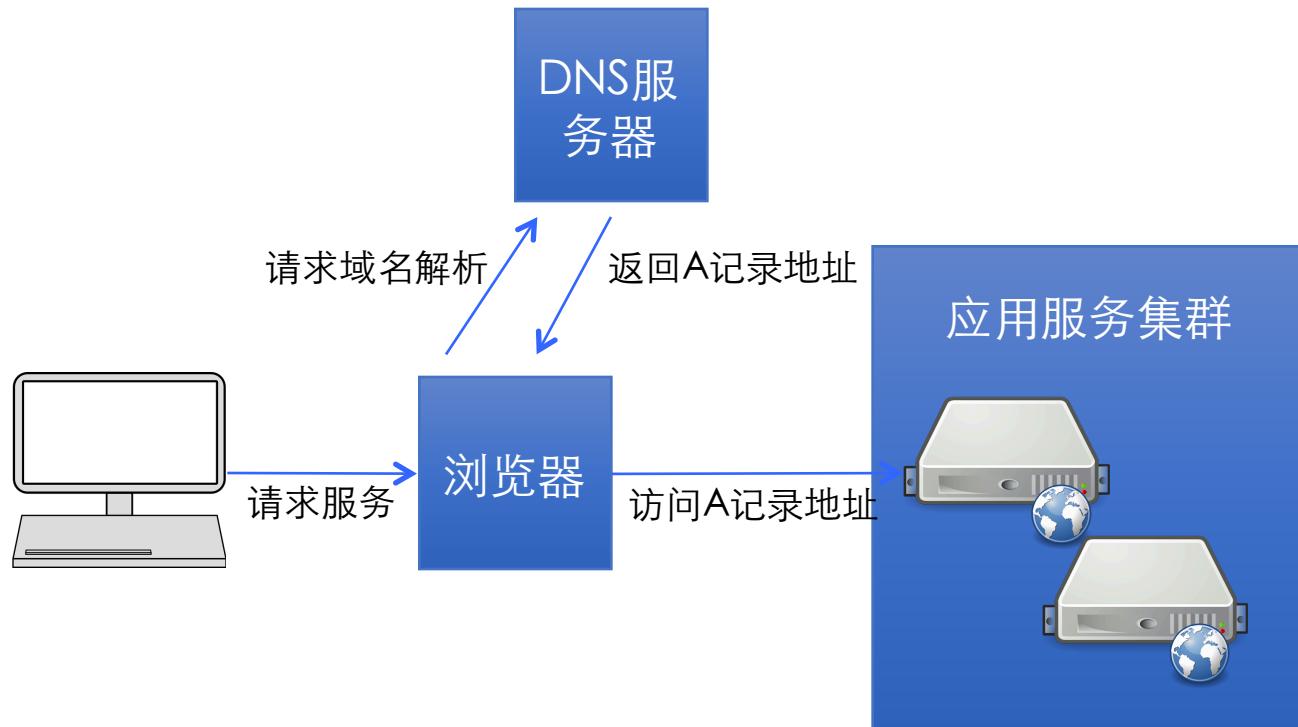
- 独立LB进程也是一种分布式方案，没有单点问题，一个LB进程挂了只影响该主机上的服务调用方。
- 服务调用方和LB之间是进程内调用，性能好。
- 简化了服务调用方，不需要为不同语言开发客户库，LB的升级不需要服务调用方改代码。
- 不足是部署较复杂，环节多，出错调试排查问题不方便。

# 负载均衡

- 系统的扩展可分为纵向（垂直）扩展和横向（水平）扩展。
  - 纵向扩展，是从单机的角度通过增加硬件处理能力，比如CPU处理能力，内存容量，磁盘等方面，实现服务器处理能力的提升，不能满足大型分布式系统（网站），大流量，高并发，海量数据的问题。
  - 横向扩展，通过添加机器来满足大型网站服务的处理能力。比如：一台机器不能满足，则增加两台或者多台机器，共同承担访问压力，这就是典型的集群和负载均衡架构。
- 负载均衡的作用（解决的问题）：
  - 解决并发压力，提高应用处理性能，增加吞吐量，加强网络处理能力；
  - 提供故障转移，实现高可用；
  - 通过添加或减少服务器数量，提供网站伸缩性，扩展性；
  - 安全防护，负载均衡设备上做一些过滤，黑白名单等处理；

# DNS负载均衡

- 最早的负载均衡技术，利用域名解析实现负载均衡，在DNS服务器，配置多个A记录，这些A记录对应的服务器构成集群。



# DNS负载均衡

## 优点

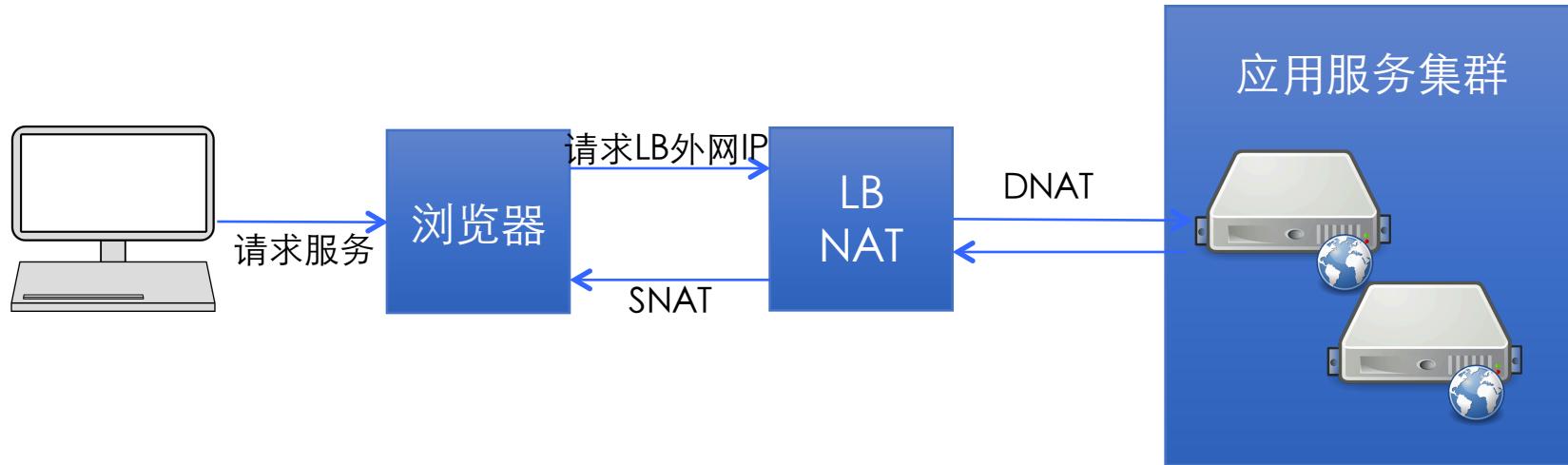
- 使用简单：负载均衡工作，交给DNS服务器处理，省掉了负载均衡服务器维护的麻烦；
- 提高性能：可以支持基于地址的域名解析，解析成距离用户最近的服务器地址，可以加快访问速度，改善性能；

## 缺点

- 可用性差：DNS解析是多级解析，新增/修改DNS后，解析时间较长；解析过程中，用户访问网站将失败；
- 扩展性低：DNS负载均衡的控制权在域名商那里，无法对其做更多的改善和扩展；
- 维护性差：也不能反映服务器的当前运行状态；支持的算法少；不能区分服务器的差异（不能根据系统与服务的状态来判断负载）

# IP负载均衡

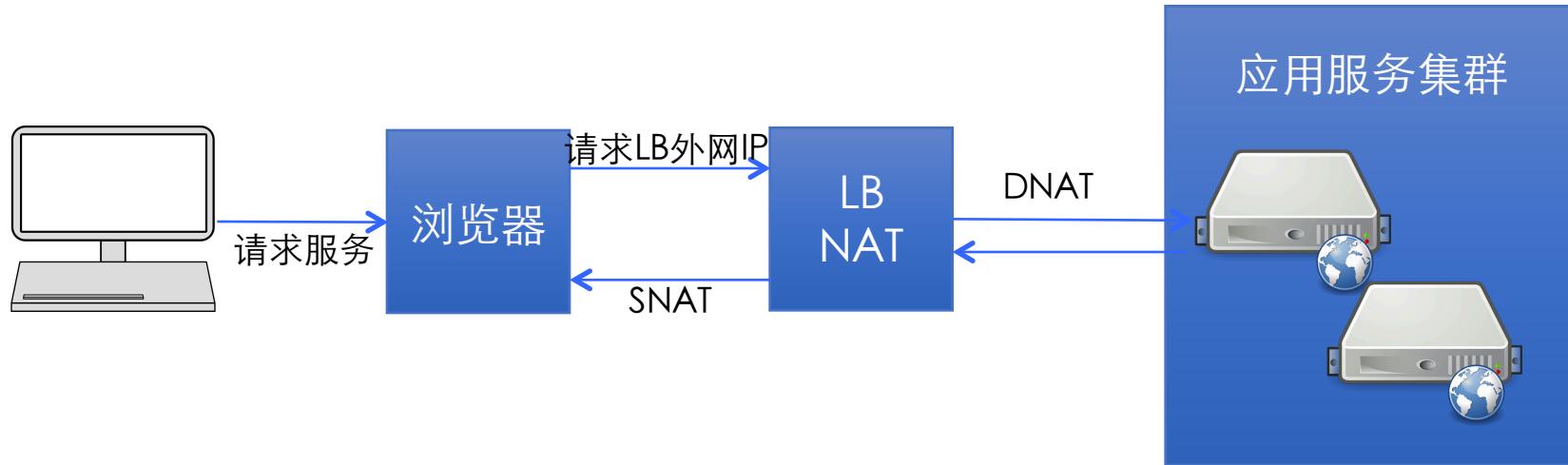
- 在网络层通过修改请求目标地址进行负载均衡。



- 用户请求数据包，到达负载均衡服务器后，负载均衡服务器在操作系统内核进程获取网络数据包，根据负载均衡算法得到一台真实服务器地址，然后将请求目的地址修改为，获得的真实ip地址，不需要经过用户进程处理。
- 真实服务器处理完成后，响应数据包回到负载均衡服务器，负载均衡服务器，再将数据包源地址修改为自身的ip地址，发送给用户浏览器。

# IP负载均衡

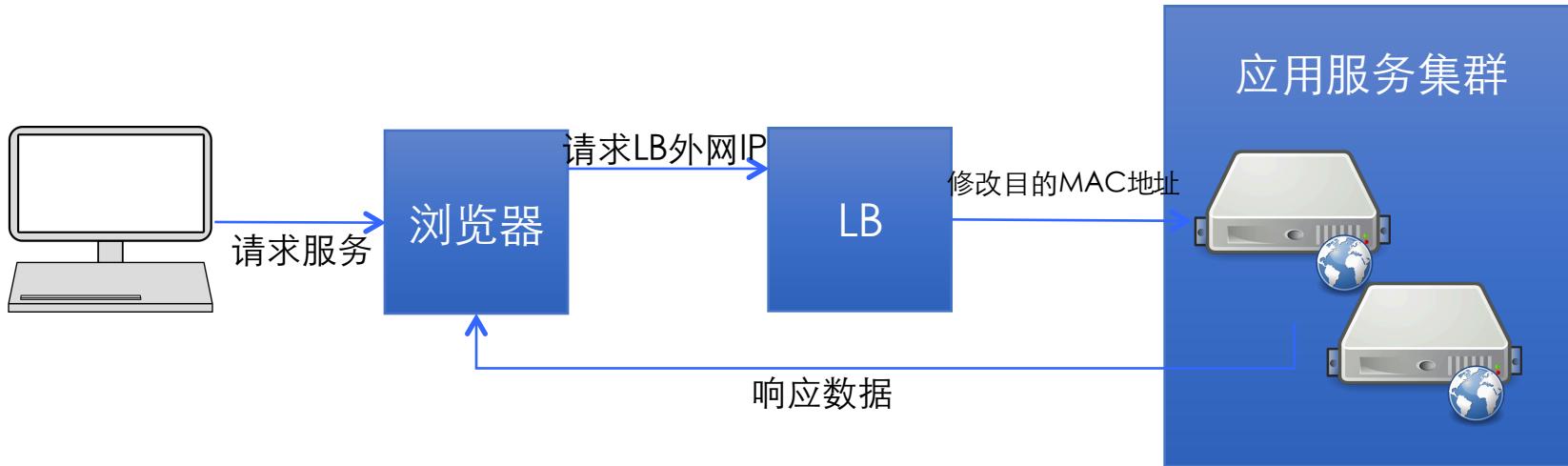
- 在网络层通过修改请求目标地址进行负载均衡。



- 优点：
  - 在内核进程完成数据分发，比在应用层分发性能更好；
- 缺点：
  - 所有请求响应都需要经过负载均衡服务器，集群最大吞吐量受限于负载均衡服务器网卡带宽；

# 链路层负载均衡

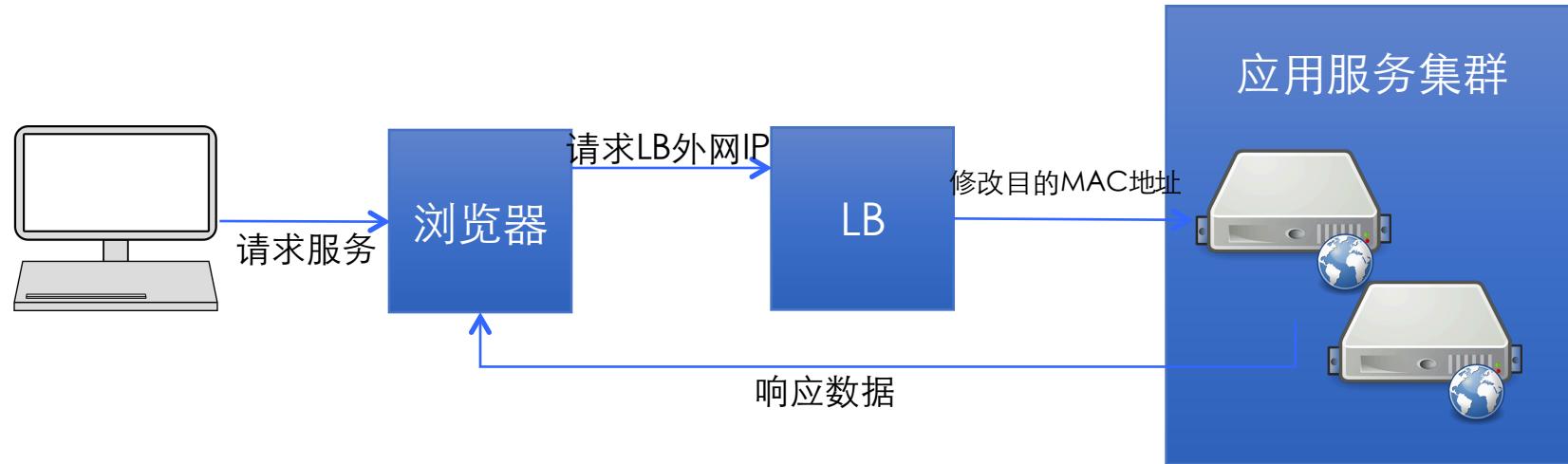
- 在通信协议的数据链路层修改mac地址进行负载均衡。



- 数据分发时，不修改ip地址，指修改目标mac地址，配置真实物理服务器集群所有机器虚拟ip和负载均衡服务器ip地址一致，达到不修改数据包的源地址和目标地址，进行数据分发的目的。
- 实际处理服务器ip和数据请求目的ip一致，不需要经过负载均衡服务器进行地址转换，可将响应数据包直接返回给用户浏览器，避免负载均衡服务器网卡带宽成为瓶颈。也称为直接路由模式（DR模式）。

# 链路层负载均衡

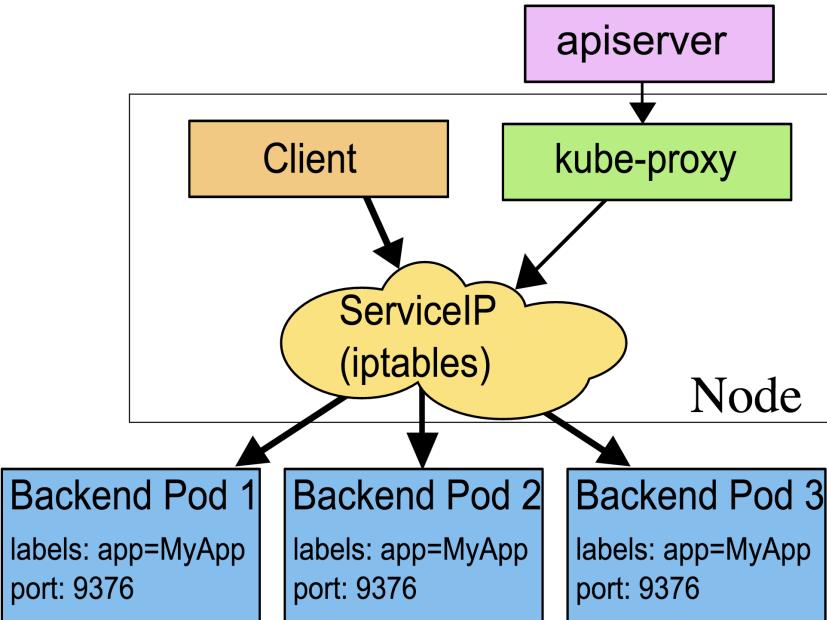
- 在通信协议的数据链路层修改mac地址进行负载均衡。



- 优点：性能好；
- 缺点：配置复杂；

# Kubernetes中的服务发现，负载均衡

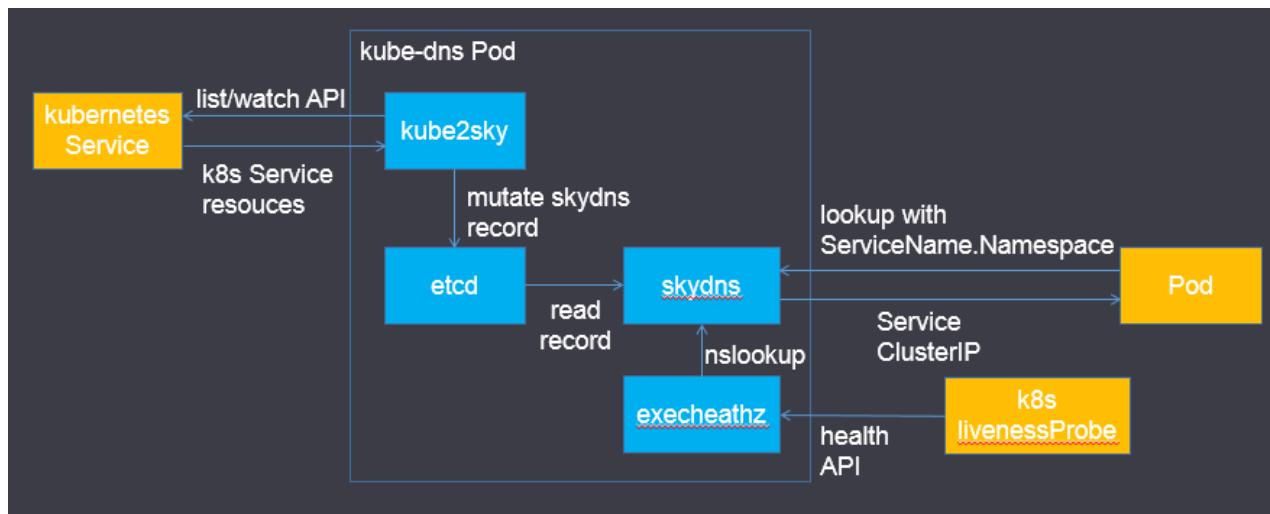
- Kubernetes在设计之初就充分考虑了针对容器的服务发现与负载均衡机制，提供了Service资源，并通过kube-proxy配合cloud provider来适应不同的应用场景。
- 在k8s集群中，服务是运行在Pod中的，在集群内可以通过获取Pod的IP来访问Pod提供的服务。但是IP并不是Durable的资源。
- 通过Service可以解决这个问题，但访问Service也需要对应的IP，因此又引入了Service发现的问题。
- Service同时能够提供集群内的L4负载均衡。



**kubernetes**  
by Google™

# Kubernetes中的服务发现，负载均衡

- 基于kube-dns插件，k8s将Service的名称当做域名注册到kube-dns中，可以通过域名来访问集群内的Service，解决了Service发现的问题。
- 为了让Pod中的容器可以使用kube-dns来解析域名，k8s会修改容器的/etc/resolv.conf配置。



**kubernetes**  
by Google™

# Kubernetes中的Ingress

- Service虽然解决了服务发现和负载均衡的问题，但它在使用上还是有一些限制：
  - 只支持4层负载均衡，没有7层功能；
  - 对外访问的时候，NodePort类型需要在外部搭建额外的负载均衡，而LoadBalancer要求kubernetes必须跑在支持的cloud provider上面；
- Ingress就是为了解决这些限制而引入的新资源，主要用来将服务暴露到cluster外面，并且可以自定义服务的访问策略。比如想要通过负载均衡器实现不同子域名到不同服务的访问。

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - backend:
              serviceName: s1
              servicePort: 80
    - host: bar.foo.com
      http:
        paths:
          - backend:
              serviceName: s2
              servicePort: 80
```



**kubernetes**  
by Google™

# 监控和日志

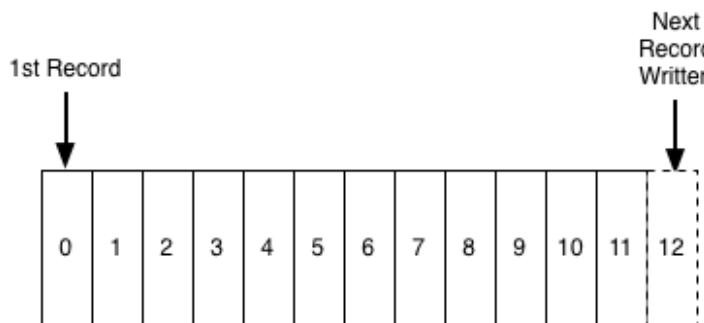
# 数据系统构建

# 数据系统构建

- 日志收集与分析
- 监控系统构建

# 日志

- 日志可能是一种最简单的不能再简单的存储抽象，只能追加、按照时间完全有序 (totally-ordered) 的记录序列。日志看起来的样子：



- 在日志的末尾添加记录，读取日志记录则从左到右。每一条记录都指定了一个唯一的顺序的日志记录编号。
- 日志记录的次序 (ordering) 定义了“时间”概念，因为位于左边的日志记录表示比右边的要早。日志记录编号可以看作是这条日志记录的“时间戳”。
- 把次序直接看成是时间概念，刚开始你会觉得有点怪异，但是这样的做法有个便利的性质：解耦了时间和任一特定的物理时钟 (physical clock)。在分布式微服务系统中，这会成为一个必不可少的性质。

# 日志

- 日志就是程序写进本地文件里的无结构的错误信息或者追踪信息？
- 在起初确实是这样的。
- 而现在由人去阅读某个机器上的日志这样的想法有些落伍过时了。当涉及很多服务和服务器时，这样的做法很快就变得难于管理，我们的目的很快就变成输入查询和输出用于理解多台机器的行为的图表。
- 另外日志也不紧紧局限于应用程序的输出：
  - 程序输出
  - 事件
  - Metrics
  - Alerts
  - Data Logs（在数据库中为了保证操作的原子性和持久性，在对数据库维护的所有各种数据结构做更改之前，数据库会把要做的更改操作的信息写入日志。）
  - Write Ahead Log（是一种数据存储方式。除了在内存中存有所有数据的状态以及节点的索引以外，通过WAL进行持久化存储。WAL中，所有的数据提交前都会事先记录日志。）

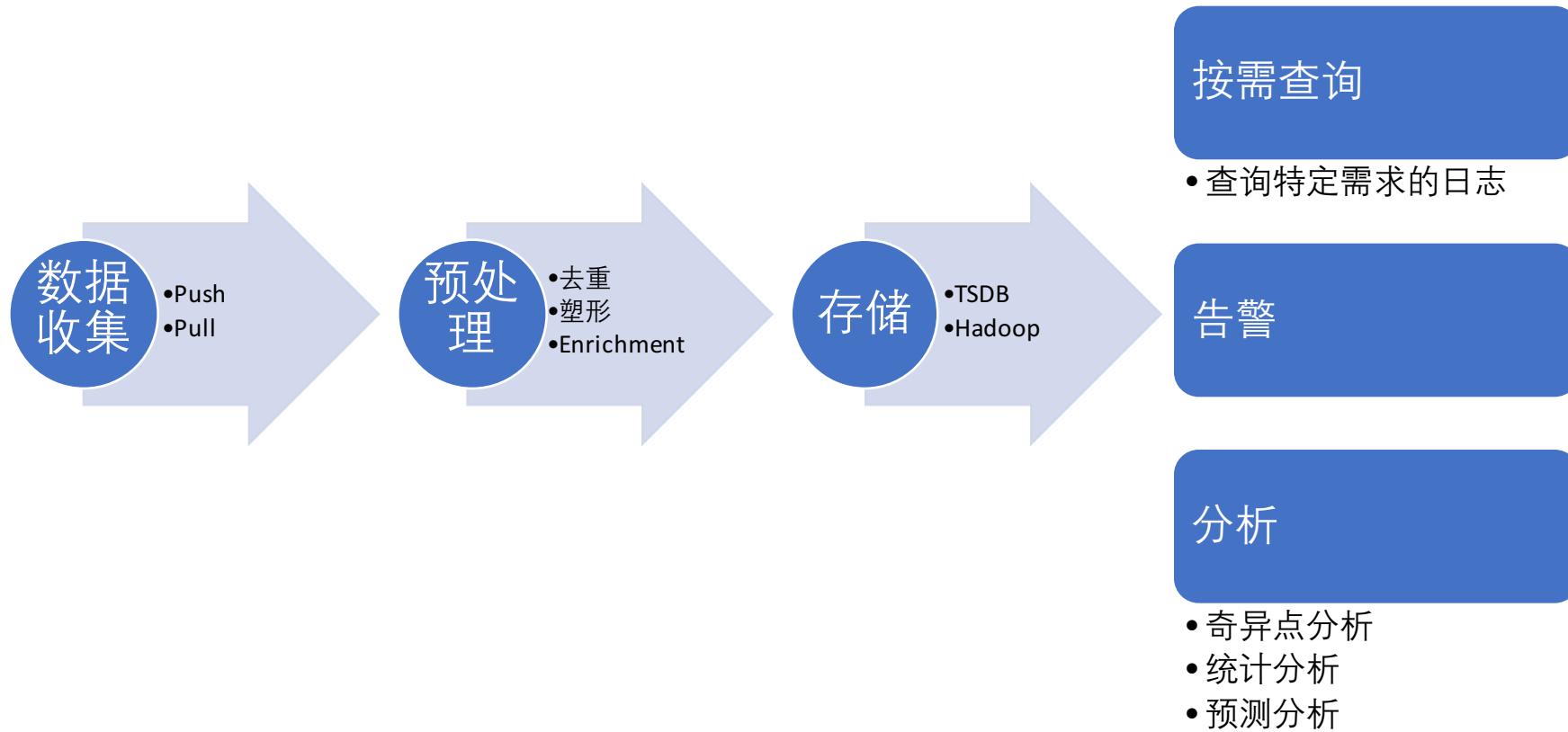
# 微服务架构下的日志

- 微服务的特点决定了功能模块的部署是分布式的，以往在单应用环境下，所有的业务都在同一个服务器上，如果服务器出现错误和异常，我们只要盯住一个点，就可以快速定位和处理问题，但是在微服务的架构下，大部分功能模块都是单独部署运行的，彼此通过总线交互，都是无状态的服务，这种架构下，前后台的业务流会经过很多个微服务的处理和传递，我们难免会遇到这样的问题：
  - 分散在各个服务器上的日志怎么处理？
  - 如果业务流出现了错误和异常，如何定位是哪个点出的问题？
  - 如何快速定位问题？
  - 如何跟踪业务流的处理顺序和结果？
- 以前在单应用下的日志监控很简单，在微服务架构下却成为了一个大问题，如果无法跟踪业务流，无法定位问题，我们将耗费大量的时间来查找和定位问题，在复杂的微服务交互关系中，我们就会非常被动。

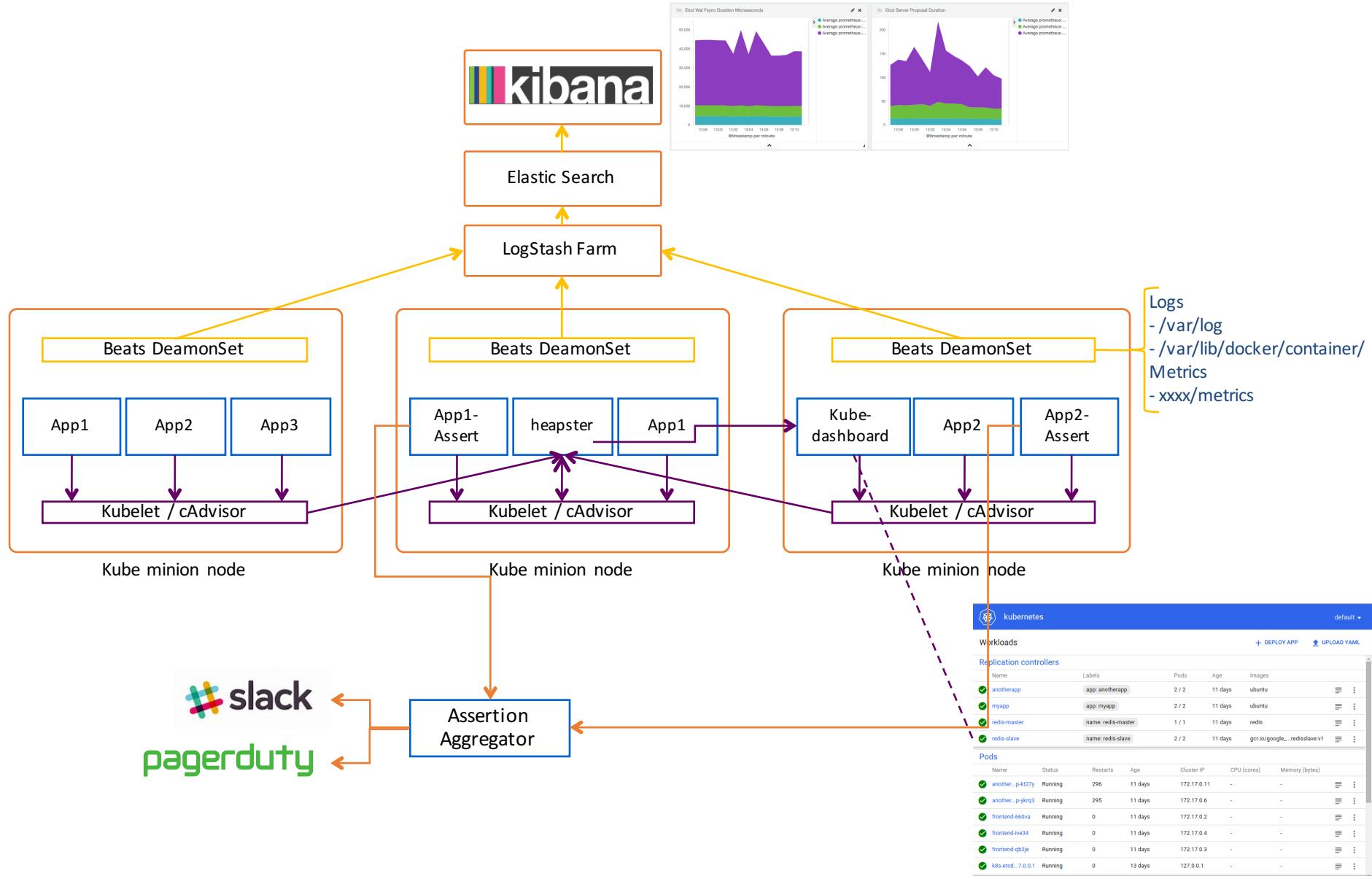
# 我们希望从日志中得到什么？

- 查看程序运行状态
- 查看出错信息
- 查看服务健康状态
- 异常告警
- 知识挖掘

# 常用数据系统构建模式



# 在Kubernetes集群中的日志系统



# 在Kubernetes集群中的日志系统

- cAdvisor

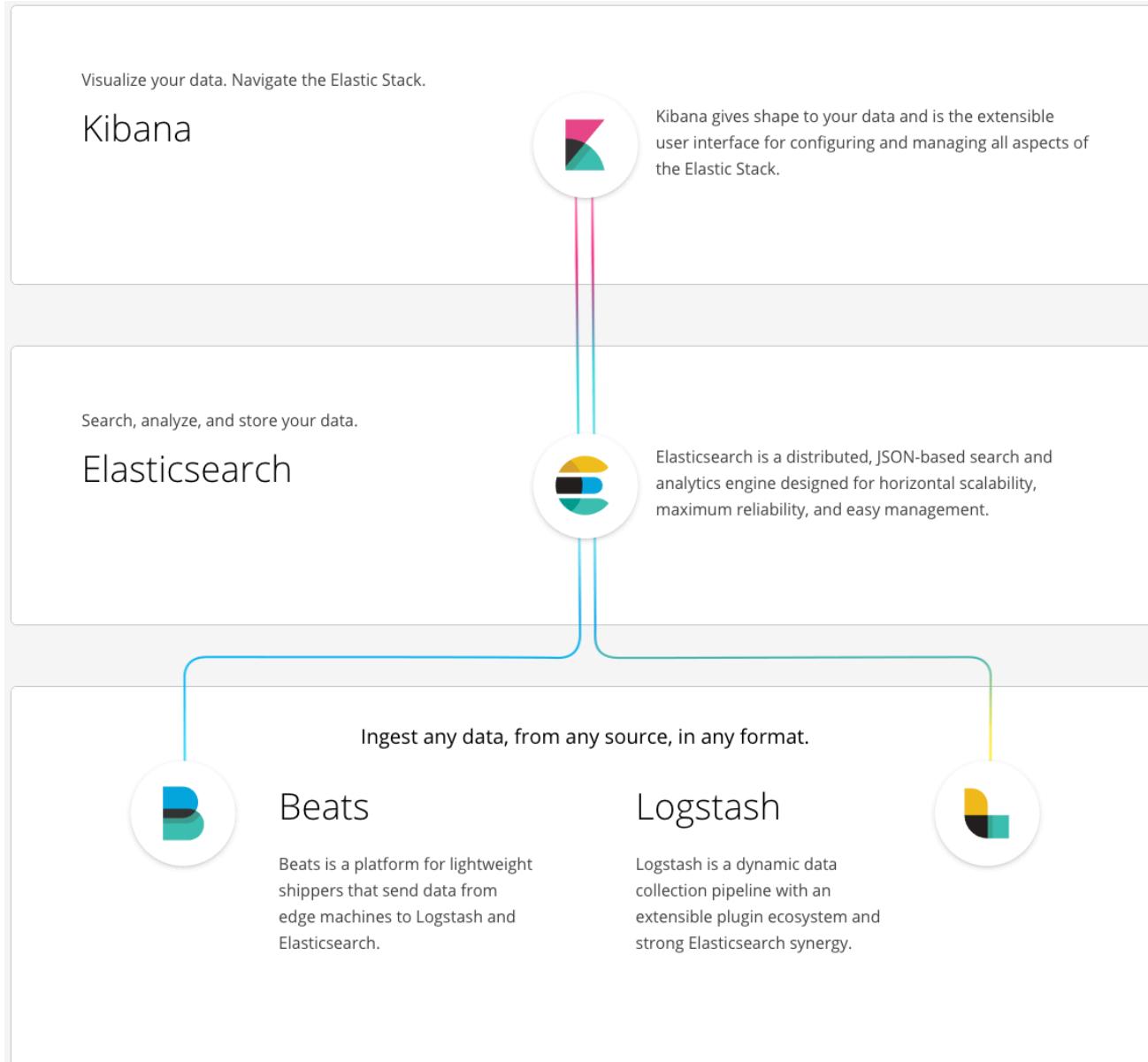
- cAdvisor is an open source container resource usage and performance analysis agent. It is purpose built for containers and supports Docker containers natively. In Kubernetes, cadvisor is integrated into the Kubelet binary. cAdvisor auto-discovers all containers in the machine and collects CPU, memory, filesystem, and network usage statistics.

- Heapster

- Heapster is a cluster-wide aggregator of monitoring and event data. It currently supports Kubernetes natively and works on all Kubernetes setups. Heapster runs as a pod in the cluster, similar to how any Kubernetes application would run. The Heapster pod discovers all nodes in the cluster and queries usage information from the nodes' Kubelets, the on-machine Kubernetes agent.



# 在Kubernetes集群中的日志系统



# 在Kubernetes集群中的日志系统



# 在Kubernetes集群中的日志系统

- Prometheus format

- Standard metrics format

```
<metric name>{<label name>=<label value>, ...}
```

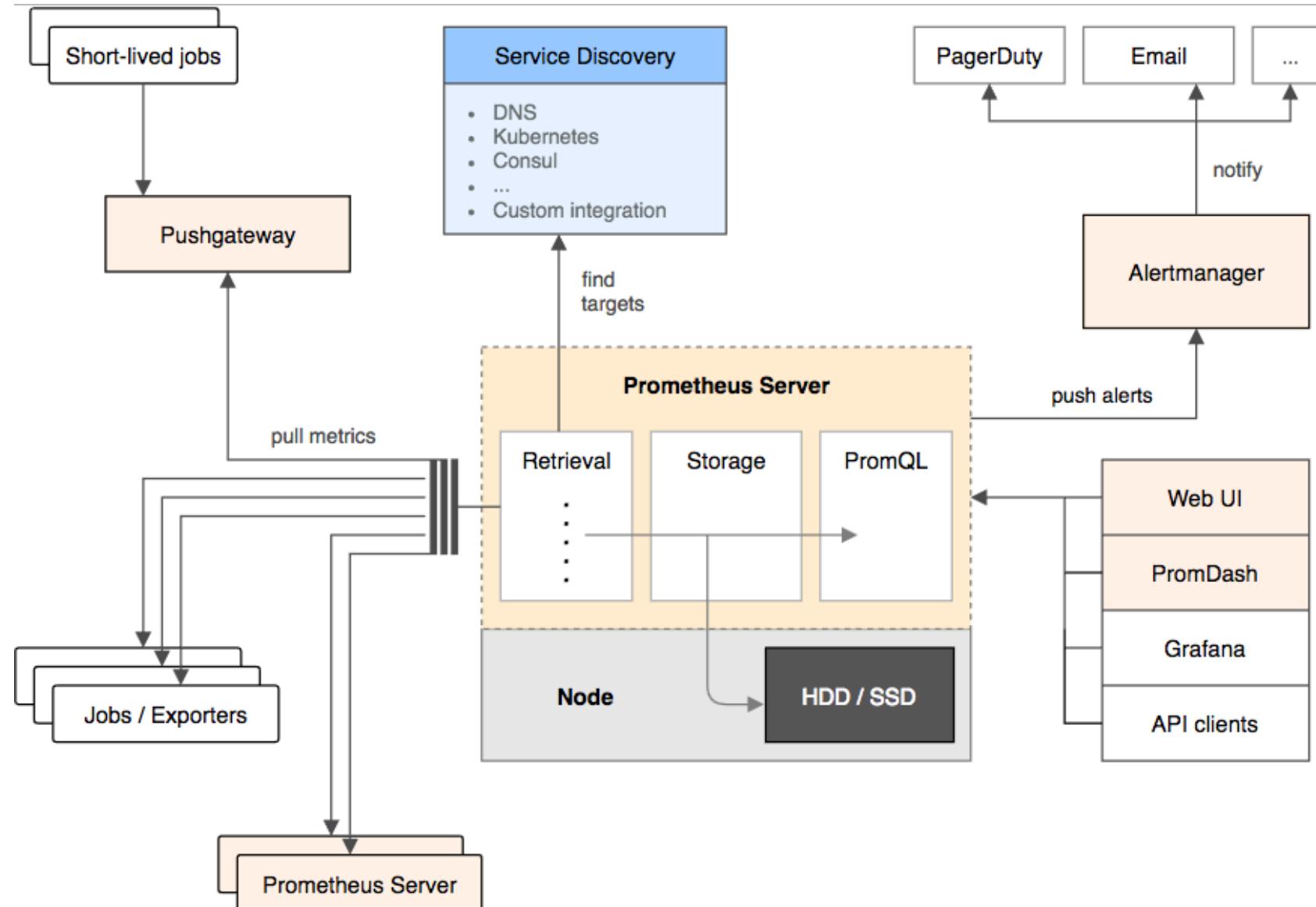
- Already integrated with kubelet

```
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="1"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="2"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="4"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="8"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="16"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="32"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="64"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="128"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="256"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="512"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="1024"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="2048"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="4096"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_bucket{le="+Inf"} 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_sum 0
etcd_debugging_mvcc_db_compaction_pause_duration_milliseconds_count 0
```

# 监控系统

- 为什么监控，监控什么内容?
  - 对自己系统的运行状态了如指掌，有问题及时发现，而不让让客户先发现我们系统不能使用。
  - 我们也需要知道我们的服务运行情况。例如，slowsql处于什么水平，平均响应时间超过200ms的占比有多少？
- 我们为什么需要监控我们的服务?
  - 需要监控工具来提醒我服务出现了故障，比如通过监控服务的负载来决定扩容或缩容。如果机器普遍负载不高，则可以考虑是否缩减一下机器规模，如果数据库连接经常维持在一个高位水平，则可以考虑一下是否可以进行拆库处理，优化一下架构。
  - 监控还可以帮助进行内部统制，尤其是对安全比较敏感的行业，比如证券银行等。比如服务器受到攻击时，我们需要分析事件，找到根本原因，识别类似攻击，发现没有发现的被攻击的系统，甚至完成取证等工作。
- 监控目的
  - 减少宕机时间
  - 扩展和性能管理
  - 资源计划
  - 识别异常事件
  - 故障排除、分析

# 在Kubernetes集群中的监控系统



# 在Kubernetes集群中的监控系统

- 每个节点的kubelet（集成了cAdvisor）会收集当前节点host上所有信息，包括cpu、内存、磁盘等。Prometheus会pull这些信息，给每个节点打上标签来区分不同的节点。

```
# HELP container_cpu_system_seconds_total Cumulative system cpu time consumed in seconds.
# TYPE container_cpu_system_seconds_total counter
container_cpu_system_seconds_total{id="/" } 735292
container_cpu_system_seconds_total{id="/system.slice"} 710067.82
container_cpu_system_seconds_total{id="/system.slice/atd.service"} 0.04
container_cpu_system_seconds_total{id="/system.slice/audittd.service"} 652.29
container_cpu_system_seconds_total{id="/system.slice/cloud-config.service"} 0
container_cpu_system_seconds_total{id="/system.slice/cloud-final.service"} 0
container_cpu_system_seconds_total{id="/system.slice/cloud-init-local.service"} 0
container_cpu_system_seconds_total{id="/system.slice/cloud-init.service"} 0
container_cpu_system_seconds_total{id="/system.slice/crond.service"} 4267.6
container_cpu_system_seconds_total{id="/system.slice/dbus.service"} 3.44
container_cpu_system_seconds_total{id="/system.slice/dm-event.service"} 428.39
container_cpu_system_seconds_total{id="/system.slice/docker.service"} 55096.24
container_cpu_system_seconds_total{id="/system.slice/dracut-shutdown.service"} 0
container_cpu_system_seconds_total{id="/system.slice/fedora-autorelabel-mark.service"} 0
container_cpu_system_seconds_total{id="/system.slice/fedora-import-state.service"} 0
container_cpu_system_seconds_total{id="/system.slice/fedora-readonly.service"} 0
container_cpu_system_seconds_total{id="/system.slice/gssproxy.service"} 27.07
container_cpu_system_seconds_total{id="/system.slice/iscsi-shutdown.service"} 0
```

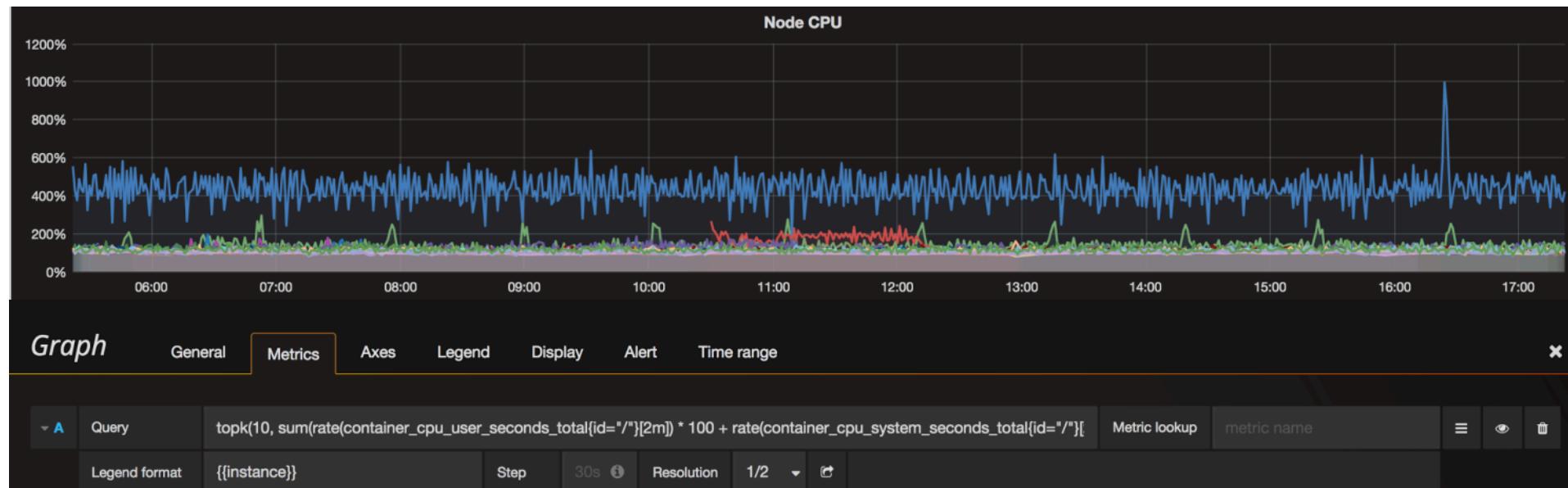
# 在Kubernetes集群中的监控系统

- Kubernetes的control panel包括各种controller都原生的暴露Prometheus格式的metrics。

```
var (
    // TODO(a-robinson): Add unit tests for the handling of these metrics once
    // the upstream library supports it.
    requestCounter = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "apiserver_request_count",
            Help: "Counter of apiserver requests broken out for each verb, API resource, client, and HTTP response contentType and code.",
        },
        []string{"verb", "resource", "client", "contentType", "code"},
    )
    requestLatencies = prometheus.NewHistogramVec(
        prometheus.HistogramOpts{
            Name: "apiserver_request_latencies",
            Help: "Response latency distribution in microseconds for each verb, resource and client.",
            // Use buckets ranging from 125 ms to 8 seconds.
            Buckets: prometheus.ExponentialBuckets(125000, 2.0, 7),
        },
        []string{"verb", "resource"},
    )
    requestLatenciesSummary = prometheus.NewSummaryVec(
        prometheus.SummaryOpts{
            Name: "apiserver_request_latencies_summary",
            Help: "Response latency summary in microseconds for each verb and resource.",
            // Make the sliding window of 1h.
            MaxAge: time.Hour,
        },
        []string{"verb", "resource"},
    )
)
```

# 在Kubernetes集群中的监控系统

- Grafana dashboard



# 运营监控界面

Tess PHX 21

PHX  
21



Tess LVS 22

LVS  
22

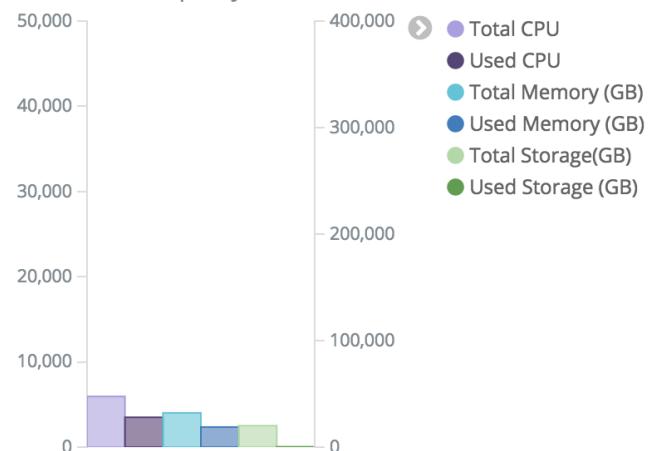


Tess SLC 23

SLC  
23



Tess PHX 21 Capacity



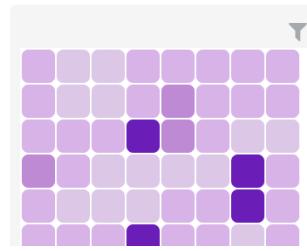
Tess PHX 21 Key Metrics

187

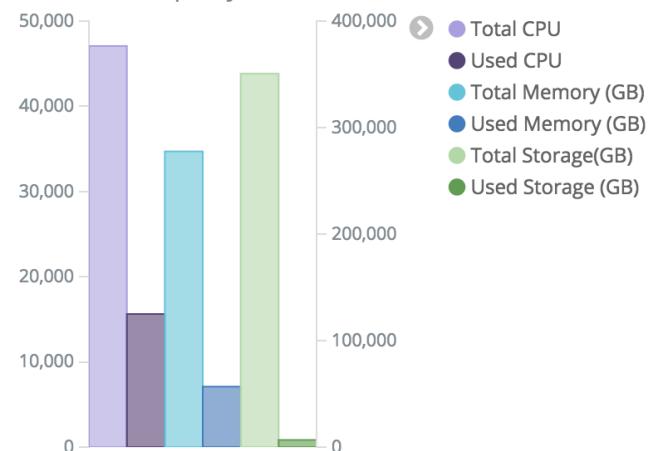
Nodes

118

Namespaces



Tess LVS 22 Capacity



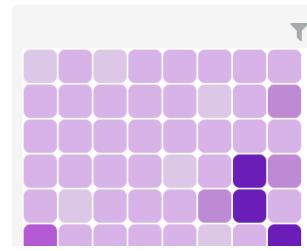
Tess LVS 22 Key Metrics

836

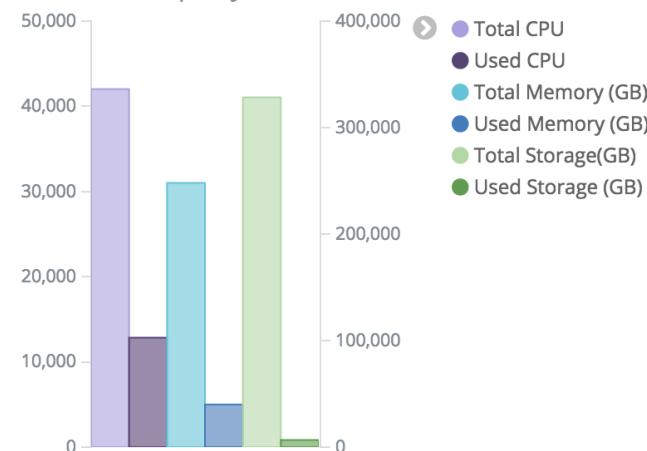
Nodes

137

Namespaces



Tess SLC 23 Capacity



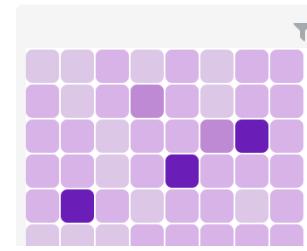
Tess SLC 23 Key Metrics

740

Nodes

100

Namespaces



Tess PHX 21 Applications

72

Applications

172

Services

Tess LVS 22 Applications

92

Applications

312

Services

Tess SLC 23 Applications

61

Applications

156

Services

# 运营监控界面

## Tess Navigation

- Global View
- Application View

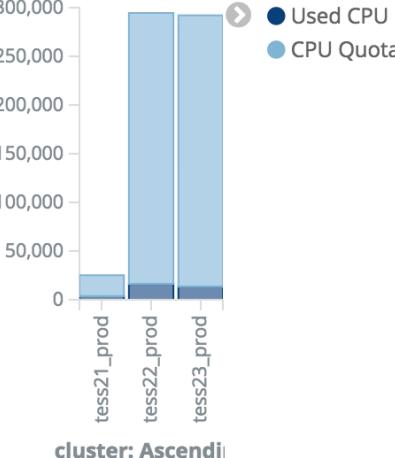
## Tess Key Metrics

**1,761** Nodes    **177** Namespaces  
**123** Applications    **435** Services  
**7,657**

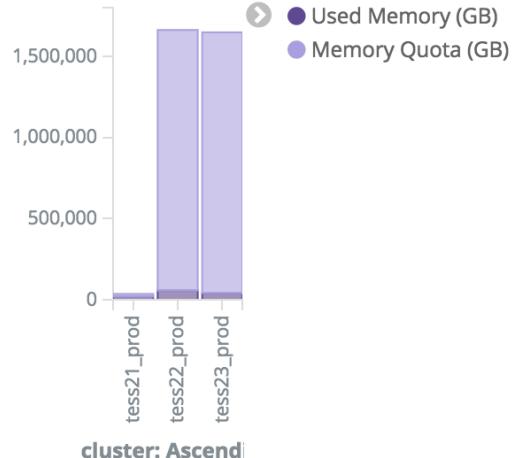
## Tess Pods By Cluster



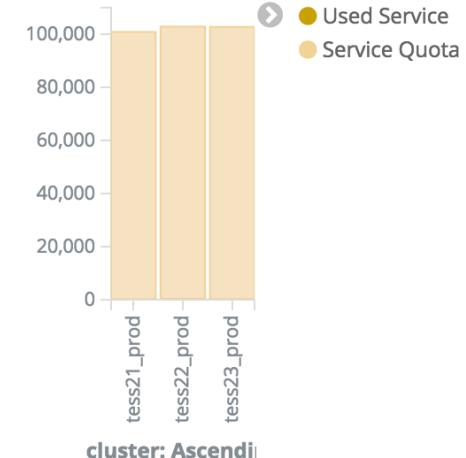
## Tess CPU Quota Usage By Cluster



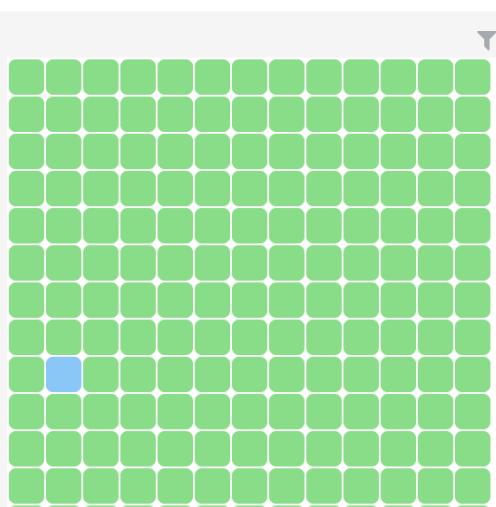
## Tess Memory Quota Usage By Cluster



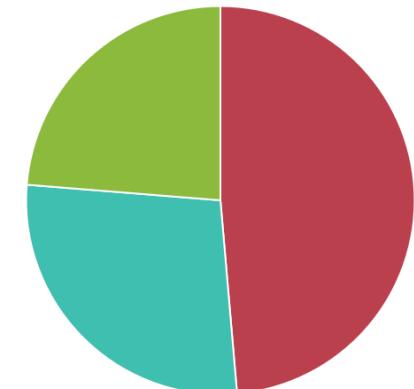
## Tess Service Quota Usage By Cluster



## Tess Pod Grid



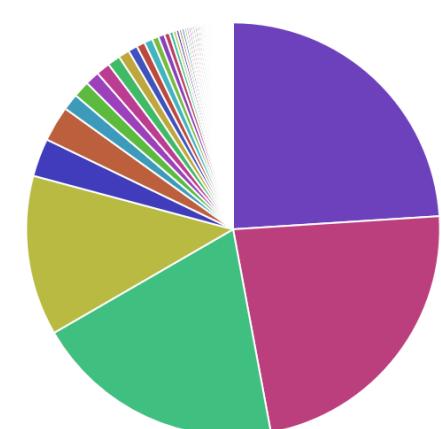
## Tess Pods By Cluster



- tess22\_prod
- tess23\_prod
- tess21\_prod

Tess Stdout Log

## Tess Pods By Namespace



- monstor
- monitoring
- kube-system
- appmon
- monstor-internal
- ebayeye
- monstor3
- ae-prod
- dyno
- default
- monstor4
- sentinel
- ceilometer-prod
- monstor-monitoring...

# 在Kubernetes集群中的告警系统

- Prometheus alert rule example

ALERT Example

IF metricsCPU > 0.9

FOR 10m

LABELS {severity="High"}

ANNOTATIONS {description="Node CPU too high: {{ \$labels.instance }} CPU {{\$value}}", summary="Node  
CPU too high"}

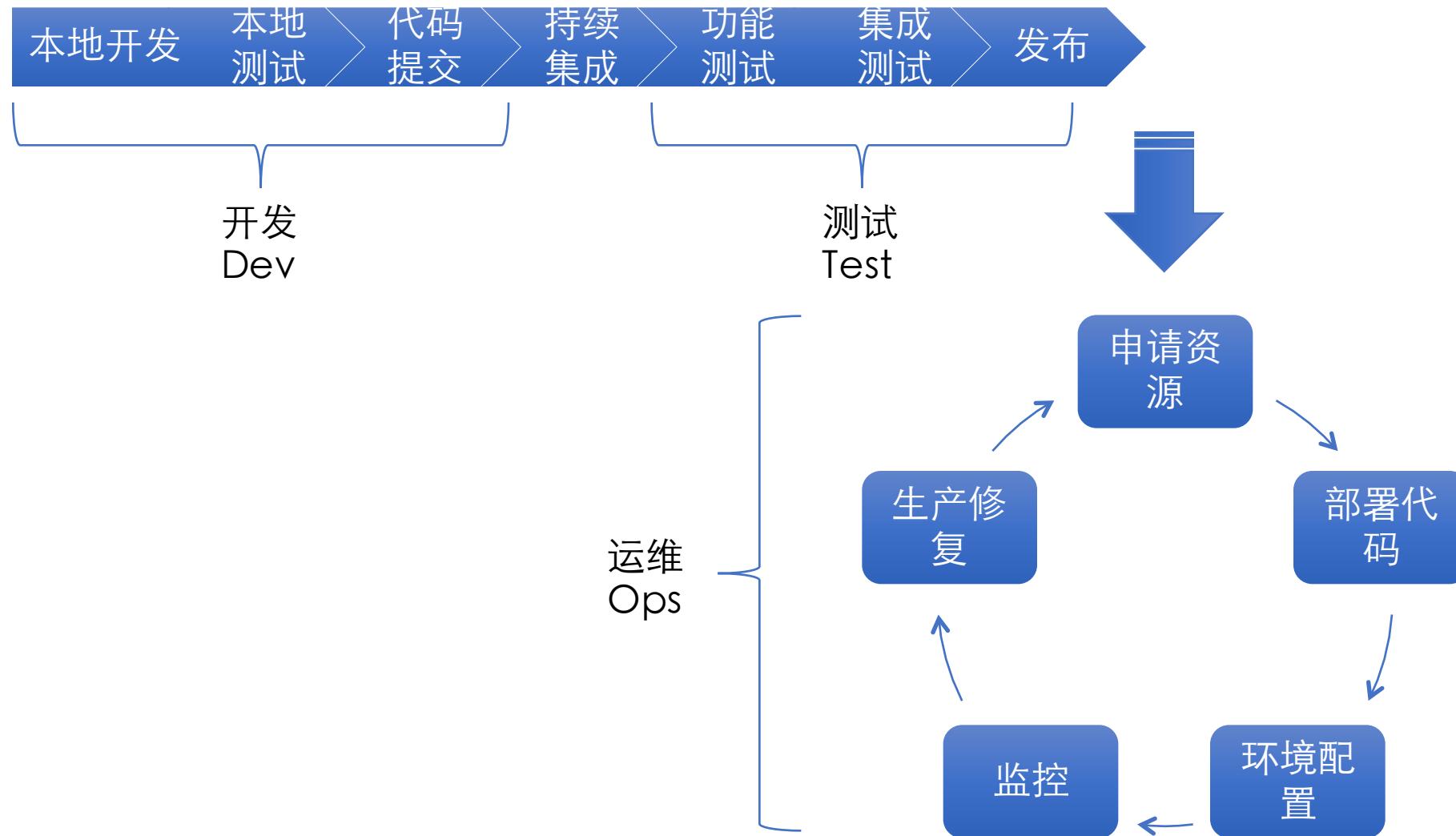
# 小结

- Kubernetes相对于其他编排系统的优势是什么?
  - 先进的设计理念
    - 打破IaaS/PaaS壁垒
    - 以应用为中心(Application Central), 对业务抽象
  - 成熟和活跃的社区
    - 最活跃的github项目
  - 项目发起人的社区领导力
    - Google
  - 生态系统和标准化
    - 生态系统:
      - Helm等上层生态应用
      - 插件化, 降低混合云门槛, 避免供应商锁定
    - 标准化, CNCF, CNI, CRI
  - 众多的商业化showcase
    - Huawei
    - Redhat
    - 众多互联网公司
      - ebay, vip

# 基于Docker的DevOps

Docker带来的最大的好处是对DevOps的变革

# DevOps 革命



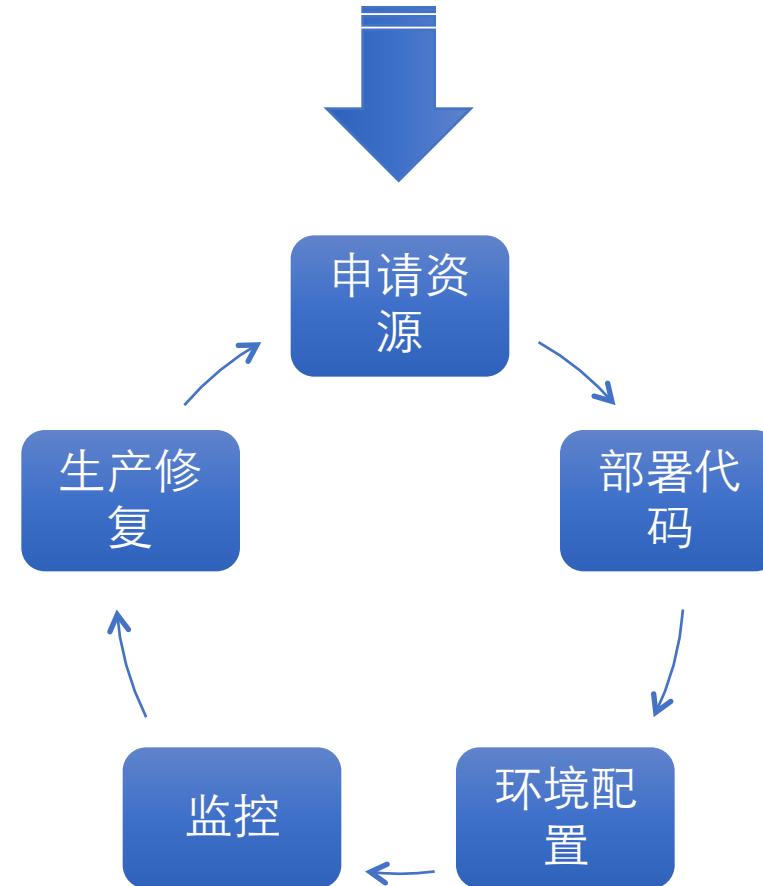
# Dev的反思

- 自己开发的代码为什么要交给别人去测试？
- 自己能不能为自己开发的代码负责？
- 线上运维为什么要用另一个团队？
- 生产环境出问题，到底谁为ATB负责？
- Dev到底知不知道真正生产环境中的痛点？
- 如何加快迭代？

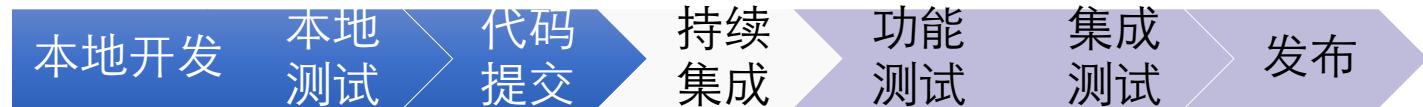
# DevOps 革命



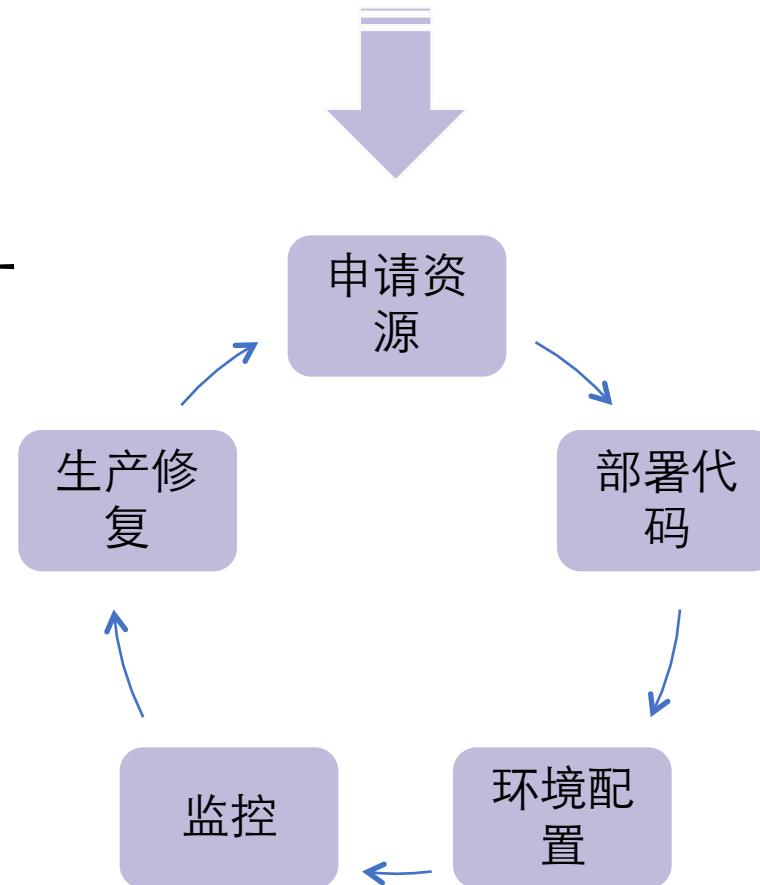
DevOps!



# Docker如何帮助DevOps?



- 开发，测试，生产环境的统一
- 自动化（配置，部署，运维）



# 开发，测试，生产环境的统一

- 开发人员为什么不愿意做运维？
- Docker官网有一句话：build once, run anywhere，容器一次构建，在其他服务器上就可以运行。
- 传统的虚拟机技术启动应用服务往往需要数分钟，而 Docker 容器应用，由于直接运行于宿主内核，无需启动完整的操作系统，因此可以做到秒级、甚至毫秒级的启动时间。大大的节约了开发、测试、部署的时间。
- 开发过程中一个常见的问题是环境一致性问题。由于开发环境、测试环境、生产环境不一致，导致有些 bug 并未在开发过程中被发现。而 Docker 的镜像提供了除内核外完整的运行时环境，确保了应用运行环境一致性，从而不会再出现“这段代码在我机器上没问题啊”这类问题。
- 开发阶段可以通过Dockerfile来进行镜像构建，并结合持续集成系统进行集成测试。运维阶段则可以直接在生产环境中快速部署该镜像，甚至结合持续部署系统进行自动部署。
- 由于Docker确保了执行环境的一致性，使得应用的迁移更加容易。Docker可以在很多平台上运行，无论是物理机、虚拟机、公有云、私有云，甚至是笔记本，其运行结果是一致的。

# 自动化（配置，部署，运维）

- Everything automation
- 自动化运维的难点在哪里?
  - 自动化需要处理的事情越多越复杂，越容易出问题。
  - 自动化过程所需要的时间越多，越容易出问题。
- Docker帮助自动化运维
  - Docker的出现革命性地改变了传统模式，部署被简化为复制和运行两个步骤。大量的部署和配置工作被提前到了编译时实现。
  - 不需要考虑运行环境差异。
  - 启动迅速。

# 经验：自动化运维是很危险的

- 谨记：往往搞垮一个数据中心的不是人，而是自动化！
- 运维自动化一定要经过标准的演变流程：

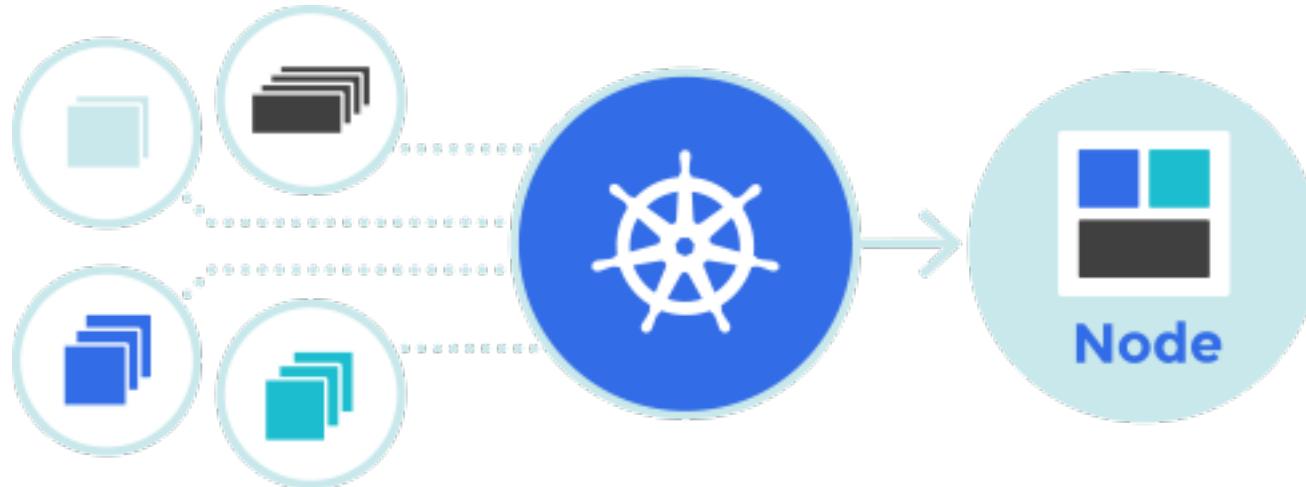


- 在设计运维自动化系统时一定要考虑一个度！也就是自动化运维一定要设计一个threshold

# 自动化运维讨论与分享



# Kubernetes能帮我们什么？



- Kubelet
- ReplicationController
- PetSet
- PersistentVolume
- Deployment
- Service & Endpoints

# Kubernetes能帮我们什么？

- Kubelet probe health check

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-healthcheck
spec:
  containers:
    - name: nginx
      image: nginx
      # defines the health checking
      livenessProbe:
        # an http probe
        httpGet:
          path: /_status/healthz
          port: 80
        # length of time to wait for a pod to initialize
        # after pod startup, before applying health checking
        initialDelaySeconds: 30
        timeoutSeconds: 1
      ports:
        - containerPort: 80
```

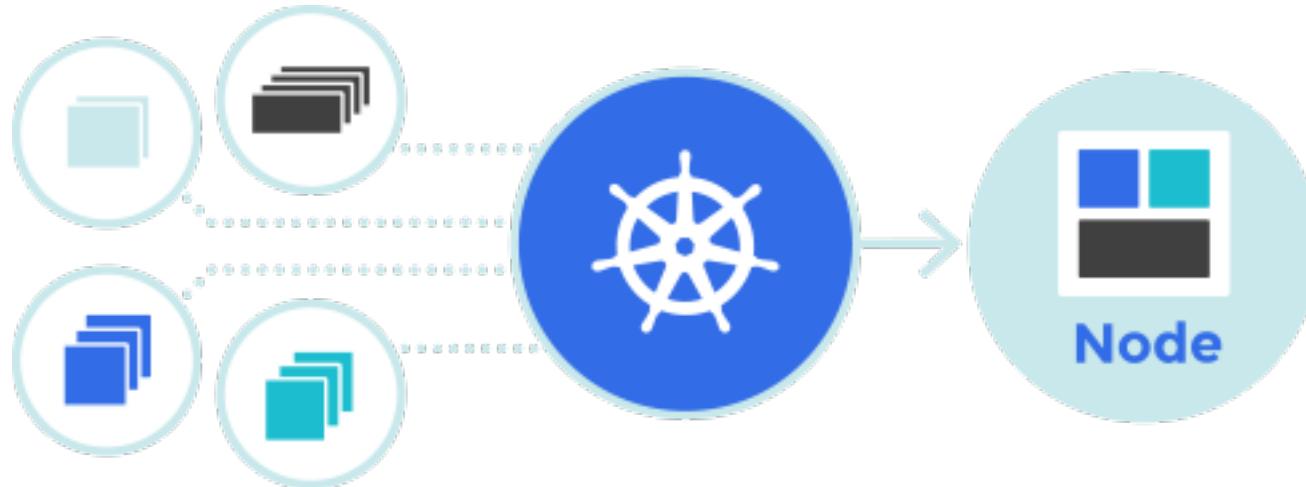
# Kubernetes能帮我们什么？

- ReplicationController & PetSet

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

```
apiVersion: apps/v1alpha1
kind: PetSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        pod.alpha.kubernetes.io/initialized: "true"
    spec:
      terminationGracePeriodSeconds: 0
    containers:
      - name: nginx
        image: gcr.io/google_containers/nginx-slim:0.8
        ports:
          - containerPort: 80
            name: web
        volumeMounts:
          - name: www
            mountPath: /usr/share/nginx/html
    volumeClaimTemplates:
      - metadata:
          name: www
          annotations:
            volume.alpha.kubernetes.io/storage-class: anything
        spec:
          accessModes: [ "ReadWriteOnce" ]
          resources:
            requests:
              storage: 1Gi
```

# Kubernetes能帮我们什么？



- Kubelet
- ReplicationController
- PetSet
- PersistentVolume
- Deployment
- Service & Endpoints

# 小结

- 容器来了，挡也挡不住
- 容器带给我们的效率
- 容器解决了DevOps中的痛点
- 容器成为自动化运维强有力的推手
- Kubernetes能帮我们减少一半的运维工作量

# 架构相关

# 微服务

- 传统IT架构面临的一些问题：
  - 使用传统的整体式架构(Monolithic Architecture)应用开发系统，如CRM、ERP等大型应用，随着新需求的不断增加，企业更新和修复大型整体式应用变得越来越困难；
  - 随着移动互联网的发展，企业被迫将其应用迁移至现代化UI界面架构以便能兼容移动设备，这要求企业能实现应用功能的快速上线；
  - 许多企业在SOA投资中得到的回报有限，SOA可以通过标准化服务接口实现能力的重用，但对于快速变化的需求，受到整体式应用的限制，有时候显得力不从心；
  - 随着应用云化的日益普及，生于云端的应用具有与传统IT不同的技术基因和开发运维模式。

# 微服务

- 从技术方面看：
  - 云计算及互联网公司大量开源轻量级技术不停涌现并日渐成熟；
  - 互联网/内联网/网络更加成熟；
  - 轻量级运行时技术的出现(node.js, WAS Liberty等)；
  - 新的方法与工具(Agile, DevOps, TDD, CI, XP, Puppet, Chef…)；
  - 新的轻量级协议(RESTful API接口, 轻量级消息机制)；
  - 简化的基础设施：操作系统虚拟化(hypervisors), 容器化(e.g. Docker), 基础设施即服务 (IaaS), 工作负载虚拟化(Kubernetes, Spark…等)等；
  - 服务平台化(PaaS)：云服务平台上具有自动缩放、工作负载管理、SLA 管理、消息机制、缓存、构建管理等各种按需使用的服务；
  - 新的可替代数据持久化模型：如NoSQL, MapReduce, BASE, CQRS等；
  - 标准化代码管理：如Github等。

这一切都催生了新的架构设计  
风格 — 微服务架构的出现

# 微服务

- 什么是微服务

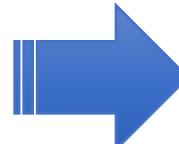
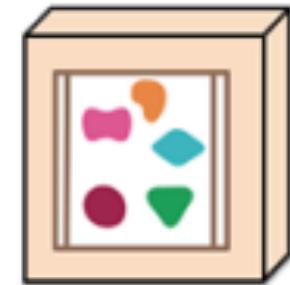
- 微服务的概念源于2014年3月Martin Fowler所写的一篇文章“Microservices” (<http://martinfowler.com/articles/microservices.html>)。
- 微服务是一种架构风格，一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力。
- 尽管“微服务”这种架构风格没有精确的定义，但其具有一些共同的特性，如围绕业务能力组织服务、自动化部署、智能端点、对语言及数据的“去集中化”控制等等。

# 微服务

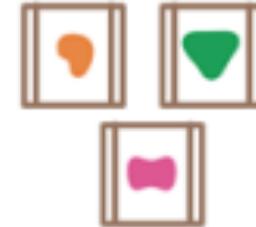
*A monolithic application puts all its functionality into a single process...*



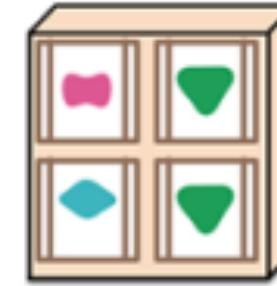
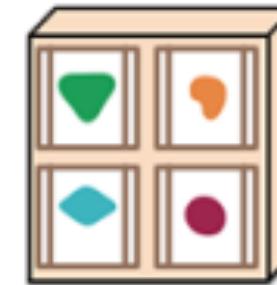
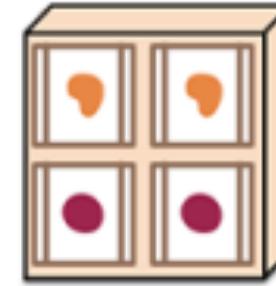
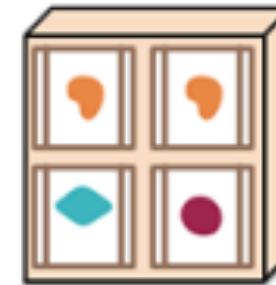
*... and scales by replicating the monolith on multiple servers*



*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by distributing these services across servers, replicating as needed.*



# 误解微服务

微服务就是API

微服务就是Web Service

微服务就是SOA

# 更全面地理解微服务

- 微服务架构的本质，是把整体的业务拆分成很多有特定明确功能的服务，通过很多分散的小服务之间的配合，去解决更大，更复杂的问题。对被拆分后的服务进行分类和管理，彼此之间使用统一的接口来进行交互。

# 更全面地理解微服务

- 通过服务实现应用的组件化(Componentization via Services)：
  - 微服务架构中将组件定义为可被独立替换和升级的软件单元，在应用架构设计中通过将整体应用切分成可独立部署及升级的微服务方式进行组件化设计。
- 围绕业务能力组织服务(Organized around Business Capabilities)：
  - 微服务架构采取以业务能力为出发点组织服务的策略，因此微服务团队的组织结构必须是跨功能的（如：既管应用，也管数据库）、强搭配的DevOps开发运维一体化团队，通常这些团队不会太大（如：亚马逊的“Two pizza team” - 不超过12人）。

# 更全面地理解微服务

- 产品而非项目模式(Products not Projects)：
  - 传统的应用模式是一个团队以项目模式开发完整的应用，开发完成后就交付给运维团队负责维护；微服务架构则倡导一个团队应该如开发产品般负责一个“微服务”完整的生命周期，倡导“谁开发，谁运营”的开发运维一体化方法。
- 智能端点与管道扁平化(Smart endpoints and dumb pipes)：
  - 微服务架构主张将组件间通讯的相关业务逻辑/智能放在组件端点侧而非放在通讯组件中，通讯机制或组件应该尽量简单及松耦合。RESTful HTTP协议和仅提供消息路由功能的轻量级异步机制是微服务架构中最常用的通讯机制。

# 更全面地理解微服务

- “去中心化” 治理(Decentralized Governance):
  - 整体式应用往往倾向于采用单一技术平台，微服务架构则鼓励使用合适的工具完成各自的任务，每个微服务可以考虑选用最佳工具完成(如不同的编程语言)。微服务的技术标准倾向于寻找其他开发者已成功验证解决类似问题的技术。
- “去中心化” 数据管理(Decentralized Data Management):
  - 微服务架构倡导采用多样性持久化(Polyglot Persistence)的方法，让每个微服务管理其自有数据库，并允许不同微服务采用不同的数据持久化技术。

# 更全面地理解微服务

- 基础设施自动化(Infrastructure Automation):
  - 云化及自动化部署等技术极大地降低了微服务构建、部署和运维的难度，通过应用持续集成和持续交付等方法有助于达到加速推出市场的目的。
- 故障处理设计(Design for failure):
  - 微服务架构所带来的一个后果是必须考虑每个服务的失败容错机制。因此，微服务非常重视建立架构及业务相关指标的实时监控和日志机制。

# 更全面地理解微服务

- 演进式的设计(Evolutionary Design):
  - 微服务应用更注重快速更新，因此系统的计会随时间不断变化及演进。微服务的设计受业务功能的生命周期等因素影响。如某应用是整体式应用，但逐渐朝微应用架构方向演进，整体式应用仍是核心，但新功能将使用应用所提供的API构建。再如在某微服务应用中，可替代性模块化设计的基本原则，在实施后发现某两个微服务经常必须同时更新，则这很可能意味着应将其合并为一个微服务。

# 微服务架构的优点

- 每个服务都比较简单，只关注于一个业务功能。
- 微服务架构方式是松耦合的，可以提供更高的灵活性。
- 微服务可通过最佳及最合适的不同编程语言与工具进行开发，能够做到有的放矢地解决针对性问题。
- 每个微服务可由不同团队独立开发，互不影响，加快推出市场的速度。
- 微服务架构是持续交付(CD)的巨大推动力，允许在频繁发布不同服务的同时保持系统其他部分的可用性和稳定性

# 但每种事物都是双刃剑

- 运维开销及成本增加：
  - 整体应用可能只需部署至一小片应用服务区集群，而微服务架构可能变成需要构建/测试/部署/运行数十个独立的服务，并可能需要支持多种语言和环境。这导致一个整体式系统如果由20个微服务组成，可能需要40~60个进程。
- 必须有坚实的DevOps开发运维一体化技能：
  - 开发人员需要熟知运维与投产环境，开发人员也需要掌握必要的数据存储技术如NoSQL，具有较强DevOps技能的人员比较稀缺，会带来招聘人才方面的挑战。
- 隐式接口及接口匹配问题：
  - 把系统分为多个协作组件后会产生新的接口，这意味着简单的交叉变化可能需要改变许多组件，并需协调一起发布。在实际环境中，一个新品发布可能被迫同时发布大量服务，由于集成点的大量增加，微服务架构会有更高的发布风险。

# 但每种事物都是双刃剑

- 代码重复：
  - 某些底层功能需要被多个服务所用，为了避免将“同步耦合引入到系统中”，有时需要向不同服务添加一些代码，这就会导致代码重复。
- 分布式系统的复杂性：
  - 作为一种分布式系统，微服务引入了复杂性和其他若干问题，例如网络延迟、容错性、消息序列化、不可靠的网络、异步机制、版本化、差异化的工作负载等，开发人员需要考虑以上的分布式系统问题。
- 异步机制：
  - 微服务往往使用异步编程、消息与并行机制，如果应用存在跨微服务的事务性处理，其实现机制会变得复杂化。
- 可测性的挑战：
  - 在动态环境下服务间的交互会产生非常微妙的行为，难以可视化及全面测试。经典微服务往往不太重视测试，更多的是通过监控发现生产环境的异常，进而快速回滚或采取其他必要的行动。但对于特别在意风险规避监管或投产环境错误会产生显著影响的场景下需要特别注意。

# 单体架构 vs 微服务架构

## 单体架构

- 整体部署
- 紧耦合
- 基于整个系统的扩展
- 集中式管理
- 应用无依赖关系管理
- 局部修改，整体更新
- 故障全局性
- 代码不易理解，难维护
- 开发效率低
- 资源利用率低
- 重，慢
- 部署简单

## 微服务架构

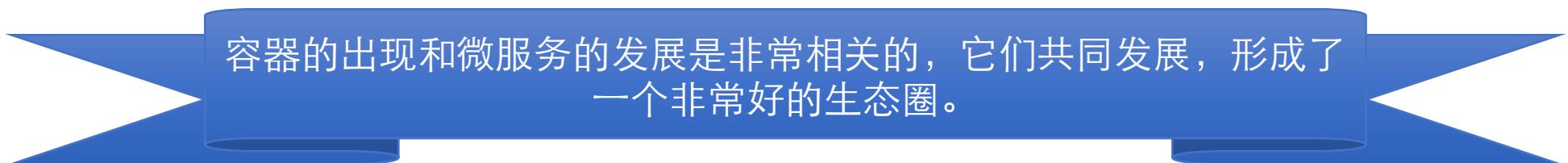
- 拆分部署
- 松耦合
- 基于独立服务，按需扩展
- 分布式管理
- 微服务间较强的依赖关系管理
- 局部修改，局部更新
- 故障隔离，非全局
- 代码易于理解维护
- 开发效率高
- 资源利用率高
- 轻，快
- 部署复杂

# 一定要避免为了“微服务”而“微服务”

- 微服务不是万能药，只有你的系统复杂度持续增加，而且业务对技术的支撑要求和预期越加强烈的时候，考虑服务化以及微服务化才是合理的，因为起码公司层面会支持你投入更多的资源和后勤保障，一家很小的创业公司，或许monolith的策略会更加合适。

# 基于容器构建微服务架构

- 使用微服务，第一步是要构建一个一体化的DevOps平台。如果你不使用DevOps做微服务的话，整个环境会变得非常的乱。它会给你的整个开发、测试和运维增加很多成本，所以第一步我们是提高DevOps的能力，能够把它的开发、部署和维护进行很完美的结合，才可以说我们真正能够享受到微服务架构的福利。
- 容器的出现给微服务提供了一个完美的环境，因为我们可以：
  - 基于容器做标准化构建和持续集成、持续交付等。
  - 基于标准工具对部署在微服务里面的容器做服务发现和管理。
  - 透过容器的编排工具对容器进行自动化的伸缩管理、自动化的运维管理。



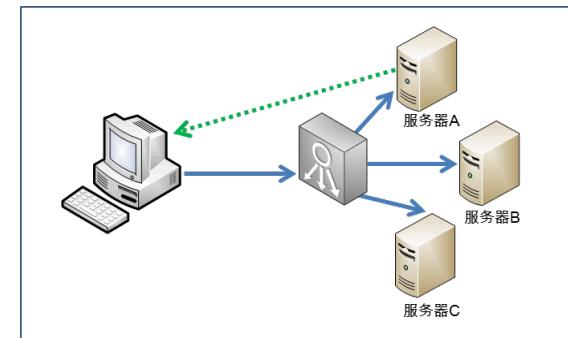
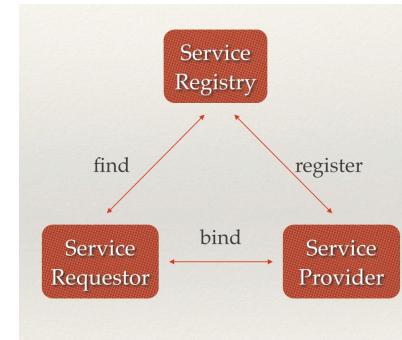
容器的出现和微服务的发展是非常相关的，它们共同发展，形成了一个非常好的生态圈。

# 微服务实施经验分享



# 实施微服务，我们需要哪些基础框架？

- 基础服务框架
- 服务注册、发现
- 负载均衡
- 健康检查



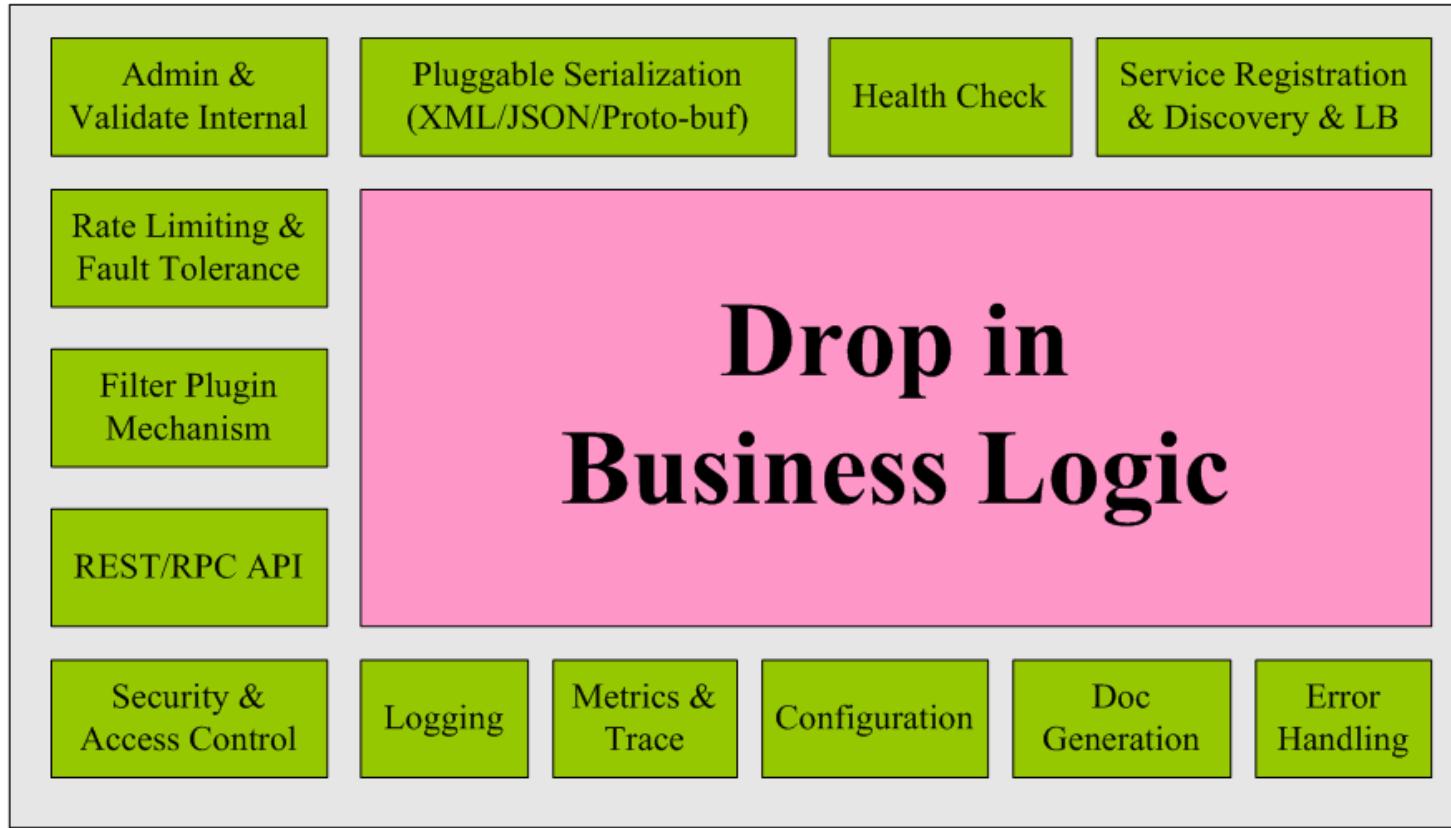
# 微服务框架

- 服务注册、发现、负载均衡和健康检查，假定采用进程内LB方案，那么服务自注册一般统一做在服务器端框架中，健康检查逻辑由具体业务服务定制，框架层提供调用健康检查逻辑的机制，服务发现和负载均衡则集成在服务客户端框架中。
- 监控日志，框架一方面要记录重要的框架层日志、metrics和调用链数据，还要将日志、metrics等接口暴露出来，让业务层能根据需要记录业务日志数据。在运行环境中，所有日志数据一般集中落地到企业后台日志系统，做进一步分析和处理。
- REST/RPC和序列化，框架层要支持将业务逻辑以HTTP/REST或者RPC方式暴露出来，HTTP/REST是当前主流API暴露方式，在性能要求高的场合则可采用Binary/RPC方式。针对当前多样化的设备类型(浏览器、普通PC、无线设备等)，框架层要支持可定制的序列化机制，例如，对浏览器，框架支持输出Ajax友好的JSON消息格式，而对无线设备上的Native App，框架支持输出性能高的Binary消息格式。

# 微服务框架

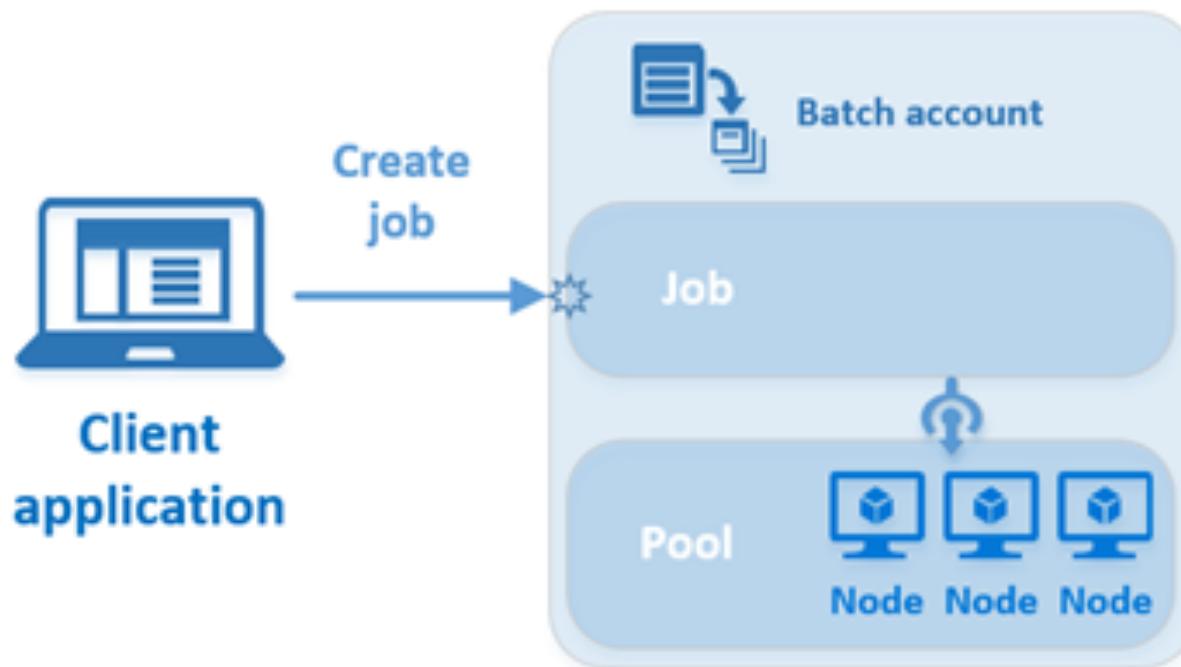
- 配置，除了支持普通配置文件方式的配置，框架层还可集成动态运行时配置，能够在运行时针对不同环境动态调整服务的参数和配置。
- 限流和容错，框架集成限流容错组件，能够在运行时自动限流和容错，保护服务，如果进一步和动态配置相结合，还可以实现动态限流和熔断。
- 管理接口，框架集成管理接口，一方面可以在线查看框架和服务内部状态，同时还可以动态调整内部状态，对调试、监控和管理能提供快速反馈。Spring Boot微框架的Actuator模块就是一个强大的管理接口。
- 统一错误处理，对于框架层和服务的内部异常，如果框架层能够统一处理并记录日志，对服务监控和快速问题定位有很大帮助。
- 安全，安全和访问控制逻辑可以在框架层统一进行封装，可做成插件形式，具体业务服务根据需要加载相关安全插件。
- 文档自动生成，文档的书写和同步一直是一个痛点，框架层如果能支持文档的自动生成和同步，会给使用API的开发和测试人员带来极大便利。Swagger是一种流行Restful API的文档方案。

# 微服务框架



- 当前业界比较成熟的微服务框架有
  - Netflix的Karyon/Ribbon
  - Spring的Spring Boot/Cloud
  - 阿里的Dubbo

# 架构设计案例：如何设计一个高可用的批处理作业系统？



# 持续集成

# 敏捷开发

- 瀑布？ 敏捷？
- 讨论： 哪个好？

# 敏捷开发

- 在IT界中，“敏捷”是一个很酷的词汇，“敏捷”的相关理论可谓铺天盖地。“敏捷”一词实质没有统一定义，各家有自家的说法。

# 敏捷开发

- 敏捷流派太多，各大门派达成了以下的一致看法，这就是“敏捷宣言”：
  - 个体和交互胜过过程和工具。
    - 以人为本，注重编程中人的自我特长发挥。
    - 可以工作的软件胜过面面具到的文档。
    - 强调软件开发的产品是软件，而不是文档。文档是为软件开发服务的，而不是开发的主体。
  - 客户合作胜过合同谈判。
    - 客户与开发者的关系是协作，不是合约。
    - 开发者不是客户业务的“专家”，也不是为了开发软件，把开发人员变成客户业务的专家。
    - 要适应客户的需求，就要通过客户合作来阐述实际的需求细节。
  - 响应变化胜过遵循计划。
    - 设计周密是为了最终软件的质量，但不表明设计比实现更重要。
    - 要适应客户需求的不断变化，设计也要不断跟进，所以设计不能是“闭门造车”、“自我良好”。
    - 要不断根据环境的变化，修改自己的设计，指导开发的方向。

# 敏捷的本质



# 经验分享

- 为了敏捷而敏捷
- 敏捷就是抛弃文档
- 敏捷就是快速上线
- Scrum实践

# 迭代

- 敏捷开发提倡将一个完整的软件版本划分为多个迭代，每个迭代实现不同的特性。重大的、优先级高的特性优先实现，风险高的特性优先实现。在项目的早期就将软件的原型开发出来，并基于这个原型在后续的迭代不断完善。迭代开发的好处是：尽早编码，尽早暴露项目的技术风险。尽早使客户见到可运行的软件，并提出优化意见。可以分阶段提早向不同的客户交付可用的版本。

# 建立持续交付的服务体系

- 传统的开发运维模式下，存在的问题：
  - 从需求到版本上线中间是个黑箱子，风险不可控；
  - 开发设计时未过多考虑运维，导致后续部署及维护的困难；
  - 开发各自为政，烟囱式开发，未考虑共享重用、联调，开发的资产积累不能快速交移到运维手中；



- 应对这样的问题，我们通常倡导的解决之道是：运维前移，统一运维，建立持续交付服务体系。

# 基于Docker的开发模式驱动持续集成

- Docker首先是一个容器级虚拟化技术，相比传统虚拟化技术，容器级的虚拟技术是操作系统内核层的虚拟，所以能节省更多资源、提升性能，意味着单位机器资源消耗下，能承载更复杂更庞大的应用系统架构。
- 启动速度更快，毫秒级的启动速度，这对于快速部署开发测试及运维环境非常有利。
- 镜像分层，部署时可以按需获取镜像生成容器并快速启动运行，因此有利于快速的部署扩容，解决运维中水平扩容的问题。
- 由于Docker镜像的天然可移植性，就像集装箱一样快速打包应用以及依赖项，在开发、测试、运维之间移动，所以可以推进开发-测试-运维环境的统一，持续集成（CI）能发挥更大的作用，例如通过CI构建应用、检查代码，打包到Docker、分发部署到测试和准生产环境，进行各类测试，都可以更方便快捷和统一。

# 基于Docker的开发模式驱动持续集成

- 开发测试环境容器化
- 持续集成容器化
- 应用交付容器化

# 开发测试环境容器化

- 容器化微服务解决的更多的是架构设计的问题，按软件工程来讲，设计之后的下一步就是开发实现的事情了，在这个阶段，传统的开发测试会有不少的问题，最突出的就是环境的问题：
  - 依赖版本不一致。
  - 软件安装麻烦、来源不一致、安装方式不一致。
  - 共用一个服务器开发环境，隔离性差，互相冲突。
  - 可移植性差，和生产环境不一致，开发人员之间也无法共享。
  - 新人入职通常又折腾一遍开发环境，无法快速搭建。

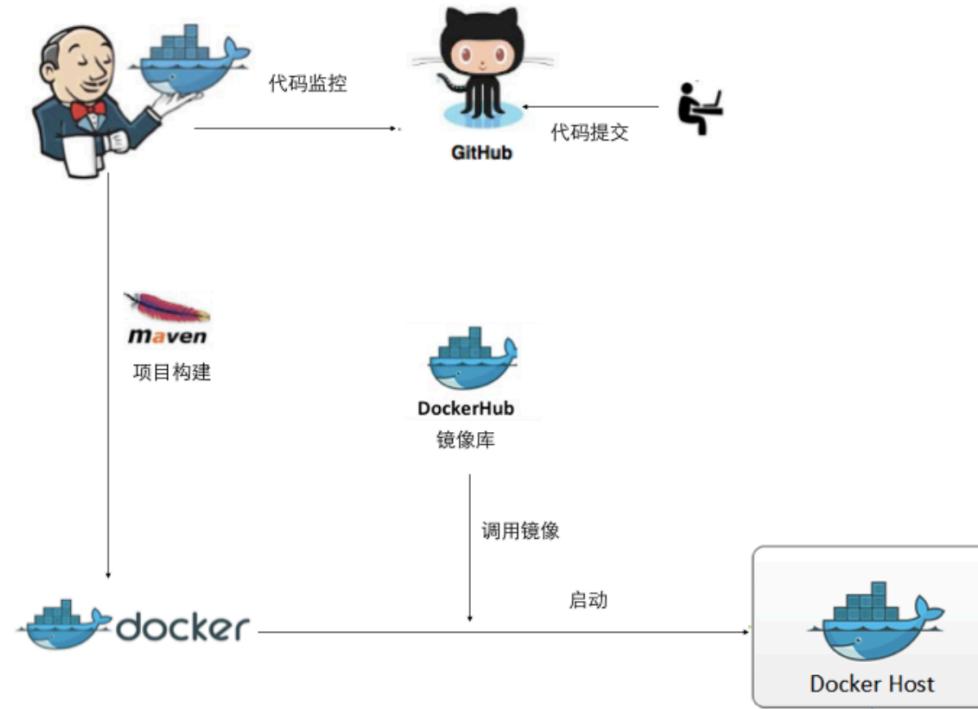
# 开发测试环境容器化

- 传统的开发模式下，有了代码库和IDE，开发人员就进行各自的开发，并在本地进行编译，测试；
- 随着开源社区的发展，目前开发一套服务需要依赖大量的组件或者代码库，甚至编译环境也有多种选择；
- 在容器化环境中，除了代码库，你还需要准备：
  - 构建环境容器化
  - 代码打包镜像化

# 实例：Kubernetes的构建

```
+++ [1229 09:39:11] Verifying Prerequisites....  
!!! [1229 09:39:12] Build image not built. Cannot clean via docker build image.  
+++ [1229 09:39:12] Removing data container kube-build-data-a9e1a6fb2f  
+++ [1229 09:39:12] Removing _output directory  
+++ [1229 09:39:12] Deleting docker image kube-build:build-a9e1a6fb2f  
+++ [1229 09:39:12] Cleaning all other untagged docker images  
+++ [1229 09:39:12] Verifying Prerequisites....  
+++ [1229 09:39:18] Building Docker image kube-build:build-a9e1a6fb2f  
+++ [1229 09:40:01] Running build command....  
+++ [1229 09:40:01] Creating data container kube-build-data-a9e1a6fb2f  
+++ [1229 09:40:19] Generating bindata:  
    /go/src/k8s.io/kubernetes/test/e2e/framework/gobindata_util.go  
+++ [1229 09:40:20] Building the toolchain targets:  
    k8s.io/kubernetes/hack/cmd/teststale  
+++ [1229 09:40:21] Building go targets for linux/amd64:  
    cmd/libs/go2idl/deepcopy-gen  
+++ [1229 09:40:27] Generating bindata:  
    /go/src/k8s.io/kubernetes/test/e2e/framework/gobindata_util.go  
+++ [1229 09:40:28] Building the toolchain targets:  
    k8s.io/kubernetes/hack/cmd/teststale  
+++ [1229 09:40:28] Building go targets for linux/amd64:  
    cmd/libs/go2idl/conversion-gen
```

# 持续集成容器化



- 最主要解决的问题是保证CI构建环境和开发构建环境的统一。
- 使用容器作为标准的构建环境，将代码库作为Volume挂载进构建容器。
- 由于构建结果是Docker镜像，所以在构建容器中执行“`docker build`”命令，需要注意DIND的问题。

# 应用交付容器化

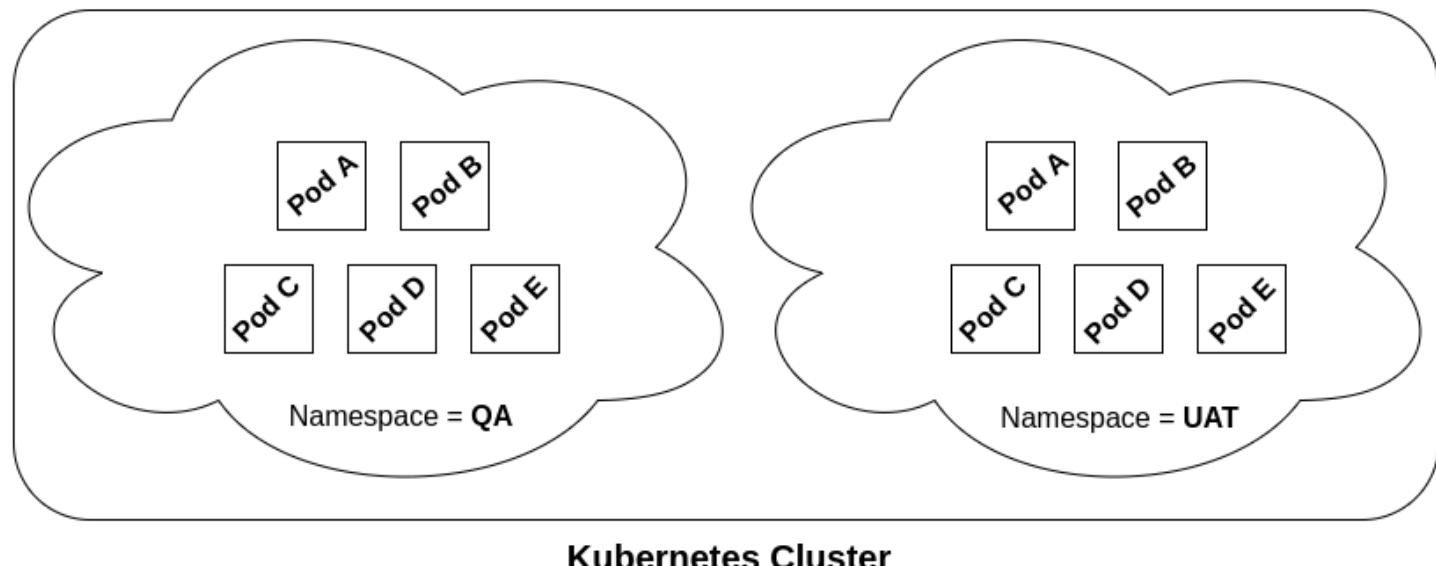
- 容器取代传统的安装包成为了微服务时代的标准交付方式。
- 容器的启动简单，标准，甚至只需要一条命令 “`docker run`”
- 尽量将所有东西打包进容器镜像中：
  - 服务依赖包
  - 服务代码
  - 服务配置数据
  - 服务启动代码

# 容器运行时管理

- Docker并没有提供容器集群管理方案，需要有额外的集群化管理与调度框架来支持。
- 当前业界主要有三种解决方案：
  - Docker Swarm。Docker官方提供的集群管理框架，使用Docker标准API，结构与功能简单，尚不适宜在生产环境使用。
  - Kubernetes。Google开源的容器管理系统，基于label和pod的概念对容器进行逻辑单元划分，实现编排与调度。功能强大，同时复杂度也较高。
  - Apache Mesos && Mesosphere Marathon。Twitter开源的老牌集群资源管理系统，主要面向数据中心资源管理，属重量级框架。

# 基于Kubernetes的测试、生产环境构建

- Kubernetes中的命名空间（namespace）提供了在一个集群中资源间的逻辑隔离。
- 提供了一种机制，使得一个集群的逻辑子集可以被赋予独立的访问控制与资源配额（resource quota）



# 基于Kubernetes的测试、生产环境构建

- 通过利用命名空间（namespace）将测试环境与生产环境进行整合：
  - 测试环境与生产环境的高度统一
  - 持续部署简单化
- 需要注意的是对不同命名空间的细粒度访问控制，以防误发布。