Master

master是集群控制节点，每个kubernetes集群里需要有一个master节点来负责整个集群的管理和控制，基本上kubernetes的所有控制命令都发给它，它来负责具体的执行过程，master节点通常会占据一个独立的服务器（高可用建议部署3台），是整个集群的大脑，如果宕机或不可用，那么对集群内容器应用的管理都将失效。

master节点上运行着一组关键进程。

- kubernetes api server（kube-apiserver）：提供了http rest 接口的关键服务进程，是kubernetes里所有资源的增、删、改、查等操作的唯一入口，也是集群控制的入口进程。
- kubernetes controller manager（kube-controller-manager）：kubernetes里所有资源对象的自动化控制中心，可以理解为资源对象的大总管。
- kubernetes scheduler（kube-scheduler）：负责资源调度（pod调度）的进程，相当于公交公司的调度室。

master节点上还需要启动一个etcd服务，因为kubernetes里的所有资源对象的数据全部是保存在etcd中的。

Node

除了master，kubernetes集群中的其他机器被称为node节点，较早的版本中称为minion。与master一样，node节点可以是一台物理机，也可以是一台虚拟机。node节点才是kubernetes集群中的工作负载节点，每个node都会被master分配一些工作负载（docker container），当某个node宕机时，其上的工作负载会被master自动转移到其他节点上去。

每个node节点上都运行着一组关键进程

- kubelet：负责pod对象的容器的创建、启停等任务，同时与master节点密切协作，实现集群管理的基本功能。
- kube-proxy：实现kubernetes service的通信与负载均衡机制的重要组件
- docker engine（docker）：docker引擎，负责本机的容器创建和管理工作。

node节点可以在运行期间动态添加到kubernetes集群中，提前时这个节点上已经正确安装、配置和启动了上述关键进行，在默认情况下kubelet会向master注册自己，这也是kubernetes推荐的node管理方式。一旦node被纳入集群管理范围，kubelet进程就会定时向master节点汇报自身的情况，，例如操作系统、docker版本、机器的cpu和内存情况，，以及当前有哪些pod在运行等，这样master可以获知每个node的资源使用情况，并实现高效均衡的资源调度策略。而某个node超过指定时间不上报信息时，会被master判定为"失联"，node的状态被标记为本可用（not ready），随后master会触发"工作负载迁移"的自动流程。

下面的命令可以查看集群中有多少个node

```
[root@vlnx251101 ~]# kubectl get nodes
NAME               STATUS    AGE       VERSION
192.168.251.101    Ready     9h        v1.6.12
192.168.251.102    Ready     9h        v1.6.12
192.168.251.103    Ready     9h        v1.6.12
```

通过 kubectl describe node <node name>来查看某个node的详细信息

```
[root@vlnx251101 ~]# kubectl describe node 192.168.251.101

#node基本信息：名称、标签、创建时间等
Name:                   192.168.251.101
Roles:                  <none>
Labels:                 beta.kubernetes.io/arch=amd64
                        beta.kubernetes.io/os=linux
                        kubernetes.io/hostname=192.168.251.101
Annotations:            node.alpha.kubernetes.io/ttl=0
```

```
                    volumes.kubernetes.io/controller-managed-attach-detach=true
CreationTimestamp:  Fri, 10 Aug 2018 20:01:10 +0800
Taints:             <none>
Unschedulable:      false
```

#node当前的运行状态，node启动以后会做一系列的自检工作，比如磁盘是否满了，如满了就标注 outofdisk=true，否则继续检查内存是否不足，memorypressure=true，
最后一切正常，就设置为ready状态，ready=true，该状态表示node处于健康状态，master将可以在其上调度新的任务（如启动pod）

```
Conditions:
  Type             Status  LastHeartbeatTime
LastTransitionTime                Reason                       Message
  ----             ------  -----------------                 ------------------
-                 ------                        -------
  OutOfDisk        False   Sat, 11 Aug 2018 16:34:39 +0800   Sat, 11 Aug 2018
00:12:29 +0800   KubeletHasSufficientDisk     kubelet has sufficient disk space
available
  MemoryPressure   False   Sat, 11 Aug 2018 16:34:39 +0800   Sat, 11 Aug 2018
00:12:29 +0800   KubeletHasSufficientMemory   kubelet has sufficient memory
available
  DiskPressure     False   Sat, 11 Aug 2018 16:34:39 +0800   Sat, 11 Aug 2018
00:12:29 +0800   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Sat, 11 Aug 2018 16:34:39 +0800   Fri, 10 Aug 2018
20:01:10 +0800   KubeletHasSufficientPID      kubelet has sufficient PID
available
  Ready            True    Sat, 11 Aug 2018 16:34:39 +0800   Sat, 11 Aug 2018
00:12:29 +0800   KubeletReady                 kubelet is posting ready status
```

#node的主机地址

```
Addresses:
  InternalIP:  192.168.251.101
  Hostname:    192.168.251.101
```

#node上资源总量：描述node可用的系统资源，包括cpu、内存数量、最大可调度pod数量等。

```
Capacity:
 cpu:                2
 ephemeral-storage:  17394Mi
```

```
 hugepages-1Gi:      0
 hugepages-2Mi:      0
 memory:             4030172Ki
 pods:               110
Allocatable:
 cpu:                2
 ephemeral-storage:  16415037823
 hugepages-1Gi:      0
 hugepages-2Mi:      0
 memory:             3927772Ki
 pods:               110
```

#主机系统信息：包括主机的唯一标识UUID、linux kernel版本号、操作系统类型与版本、kubernetes版本号、kubelet与kube-proxy的版本号等

```
System Info:
 Machine ID:                 c1f88cf1d72c4cd2928071ece8e4fe1e
 System UUID:                23224D56-9493-7053-65FB-7017C7526D85
 Boot ID:                    0f4af91d-a0fa-44d8-9bb5-36d828876ff4
 Kernel Version:             3.10.0-693.2.2.el7.x86_64
 OS Image:                   CentOS Linux 7 (Core)
 Operating System:           linux
 Architecture:               amd64
 Container Runtime Version:  docker://1.13.1
 Kubelet Version:            v1.11.2
 Kube-Proxy Version:         v1.11.2
```

#当前正在运行的pod列表概要信息

```
Non-terminated Pods:         (6 in total)
  Namespace                  Name                                CPU
Requests  CPU Limits  Memory Requests  Memory Limits
  ---------                  ----                                -----------
-  ----------  ---------------  -------------
  default                    my-nginx-59497d7745-gqns8           0
(0%)        0 (0%)      0 (0%)           0 (0%)
  default                    nginx                               0
(0%)        0 (0%)      0 (0%)           0 (0%)
  kube-system                heapster-55884f49b6-zm2jv           0
(0%)        0 (0%)      0 (0%)           0 (0%)
```

```
  kube-system                 kube-dns-8498694664-7rbm6                 260m
(13%)     0 (0%)       110Mi (2%)       170Mi (4%)
  kube-system                 monitoring-grafana-84fd47f8c9-ghg7k      0
(0%)         0 (0%)       0 (0%)            0 (0%)
  kube-system                 monitoring-influxdb-64b7644788-ggp49     0
(0%)         0 (0%)       0 (0%)            0 (0%)


#已分配的资源使用概要信息，例如资源申请的最低、最大允许使用量占系统总量的百分比
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource  Requests    Limits
  --------  --------    ------
  cpu       460m (23%)  1100m (55%)
  memory    110Mi (2%)  170Mi (4%)

#node相关的event信息
Events:     <none>
```

Node管理

禁止pod调度到该节点上

```
kubectl cordon <node>
```

驱逐该节点上的所有pod

```
kubectl drain <node>
```

该命令会删除该节点上的所有Pod（DaemonSet除外），在其他node上重新启动它们，通常该节点需要维护时使用该命令。直接使用该命令会自动调用`kubectl cordon <node>`命令。当该节点维护完成，启动了kubelet后，再使用`kubectl uncordon <node>`即可将该节点添加到kubernetes集群中。

Namespace

Namespace（命名空间）是 Kubernetes 系统中的另一个非常重要的概念，Namespace 在很多情况下用于实现多租户的资源隔离。Namespace 通过将集群内部的资源对象“分配”到不同的 Namespace 中，形成逻辑上分组的不同项目、小组或用户组，便于不同的分组在共享使用整个集群的资源的同时还能被分别管理。

Kubernetes 集群在启动后，会创建一个名为“default”的 Namespace，通过 kubectl 可以查看到：

```
[root@vlnx251101 ~]# kubectl get namespace
NAME          STATUS    AGE
default       Active    1d
kube-public   Active    1d
kube-system   Active    1d
```

接下来，如果不特别指明 Namespace，则用户创建的 Pod、RC、Service 都将被系统创建到这个默认的名为 default 的 Namespace 中。

Namespace 的定义很简单。如果所示的 yaml 定义名为 development 的 Namespace。

```
[root@vlnx251101 ~]# vim development-ns.yaml

apiVersion: v1
kind: Namespace
metadata:
  name: development
```

一旦创建了 Namespace，在创建资源对象时就可以指定这个资源对象属于哪个 Namespace。比如下面的例子中，定义了一个名为 busybox 的 Pod，放入 development 这个 Namespace 里：

```
[root@vlnx251101 ~]# vim busybox-pod.yaml

apiVersion: v1
kind: Pod
metadata:
```

```
  name: busybox
  namespace: development
spec:
  containers:
    - image: busybox
      command:
        - sleep
        - "3600"
      name: bosybox
```

创建出这个 Namespace 和 Pod 服务：

```
[root@vlnx251101 ~]# kubectl apply -f development-ns.yaml -f busybox-pod.yaml
namespace "development" created
pod "busybox" created
```

此时，使用 get 命令查看将无法显示：

```
[root@vlnx251101 ~]# kubectl get pod
NAME                                READY     STATUS    RESTARTS   AGE
mysql-nt7ck                         1/1       Running   0          1d
myweb-rl28l                         1/1       Running   0          1d
myweb-tlpjr                         1/1       Running   0          1d
tomcat-deployment-1473713186-6kg7m  1/1       Running   0          2h
```

这是因为如果不加参数，则 kubectl get 命令将仅显示属于 default namespace 的资源对象。

可以在 kubectl 命令中加入 --namespace 参数来查看某个命名空间中的对象：

```
[root@vlnx251101 ~]# kubectl get pods --namespace=development
NAME      READY     STATUS         RESTARTS   AGE
busybox   0/1       ErrImagePull   0          26s
```

或者显示全部 Namespace Pod：

```
[root@vlnx251101 ~]# kubectl get pod --all-namespaces
NAMESPACE     NAME                                READY
STATUS              RESTARTS   AGE
default       mysql-nt7ck                         1/1
Running             0          1d
default       myweb-rl28l                         1/1
Running             0          1d
```

```
default       myweb-tlpjr                              1/1
Running             0          1d
default       tomcat-deployment-1473713186-6kg7m   1/1
Running             0          2h
development   busybox                                  0/1
ImagePullBackOff    0          52s
```

当给每个租户创建一个 Namespace 来实现多租户的资源隔离时，还能结合 Kubernetes 的资源配合管理，限定不同租户能占用的资源，例如 CPU 使用量、内存使用量等。

Label

Label 是 Kubernetes 中的核心概念。一个 Label 是一个 key=value 的键值对，其中 key 与 value 由用户自己制定。Label 可以附加到各种资源对象，例如 Node、Pod Service、RC 等，一个资源对象可以定义任意数量的 Label，同一个 Label 也可以被添加到任意数量的资源对象上去，Label 通常在资源对象定义时确定，也可以在对象创建后动态添加或删除。

可以通过给指定的资源对象捆绑一个或多个不同的 Label 来实现多维度的资源分组管理功能，以便于灵活、方便地进行资源分配、调度、配置、部署等管理工作。例如：部署不同版本的应用到不同的环境中；或者监控和分析应用（日志记录、监控、告警）等。一些常用的 Label 示例如下。

- 版本标签："release": "stable"，"release"："canary"...
- 环境标签："environment"："dev"，"environment"："qa"，"environment"："oriduction"
- 架构标签："tier"："frontend"，"tier"："backend"，"tier"："middleware"
- 分区标签："partition"："customerA"，"partition"："customerB".

..

- 质量管控标签："track"："daily"，"track"："weekly"

Label 相当于我们熟悉的“标签”，给某个资源对象定义一个 Label，就相当于给它打了一个标签，随后可以通过 Label Selector（标签选择器）查询和筛选拥有某些 Label 的资源对象，Kubernetes 通过这种方式实现了类似 SQL 的简单又通用的对象查询机制。

Label Selector 可以被类比为 SQL 语句中的 where 查询条件，例如，name=redis-slave 这个 Label Selector 作用于 Pod 时，可以被类比为 SELECT * FROM pod WHERE pods name = 'redis-slave' 这样的语句。当前有两种 Label Selector 的表达式：基于等式的（Equality-based）和基于集合的（Set-based），前者采用“等式类”的表达式匹配标签，下面是一些具体的例子。

- name = redis-slave：匹配所有具有标签 name = redis-slave 的资源对象。
- env != production：匹配所有不具有标签 env=production 的资源对象，比如 env=dev 就是满足此条件的标签之一。

而后者则使用集合操作的表达式匹配标签，下面是一些具体的例子。

- name in（redis-master，redis-slave）：匹配所有具有标签 name=redis-master 或者 name=redis-slave 的资源对象。
- name not in（php-frontend）：匹配所哟不具有标签 name=php-frontend 的资源对象。

可以通过多个 Label Selector 表达式的组合实现复杂的条件选择，多个表达式之间用“,”进行分割即可，几个条件之间是“AND”的关系，即同时满足多个条件，比如下面的例子：

```
name=redis-slave,env!=production
name not in (php-frontend),env!=production
```

以 myweb Pod 为例，Label 定义在其 metadata 中：

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: myweb
  labels:
    app: myweb
```

管理对象 RC 和 Service 在 spec 中定义 Selector 与 Pod 进行关联：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myweb
spec:
  replicas: 1
  selector:
    app: myweb
  template:
...
apiVersion: v1
kind: Service
metadata:
  name: myweb
spec:
  selector:
    app: myweb
  ports:
  - port: 8080

[root@vlnx251101 test]# cat <<EOF > a.yaml
apiVersion: v1
kind: Pod
metadata:
  name: myweb
  labels:
```

```
    app: myweb
spec:
  containers:
    - image: busybox
      command:
        - sleep
        - "3600"
      name: bosybox
EOF

[root@vlnx251101 test]# kubectl create -f a.yaml


[root@vlnx251101 test]# kubectl get pods -l app=myweb
NAME       READY     STATUS    RESTARTS   AGE
myweb      1/1       Running   0          2m
```

新出现的管理对象如 Deployment、ReplicaSet、DaemonSet 和 Job 则可以在 Selector 中使用基于集合的筛选条件定义，例如：

```
selector:
  matchLabels:
    app: myweb
  matchExpressions:
    - {key: tier, operator: In, values: [frontend]}
    - {key: environment, operator: NotIn, values: [dev]}
```

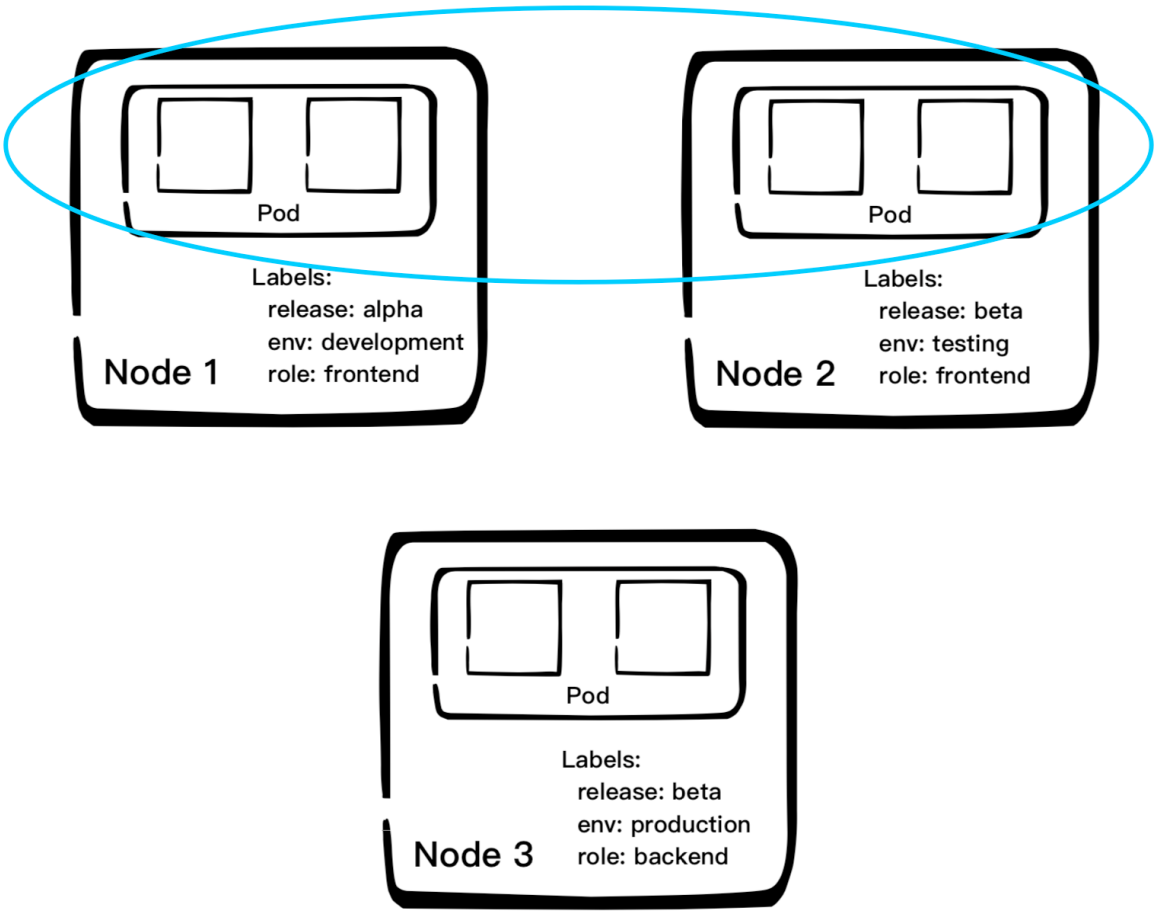
matchLabels 用于定义一组 Label，与直接写在 Selector 中作用相同；matchExpressions 用于定义一组基于集合的筛选条件，可用的条件运算包括：In、NotIn、Exists 和 DoesNotExist。

如果同时设置了 matchLabels 和 matchExpressions，则两组条件为“AND”关系，及所有条件需要同时满足才能完成 Selector 筛选。

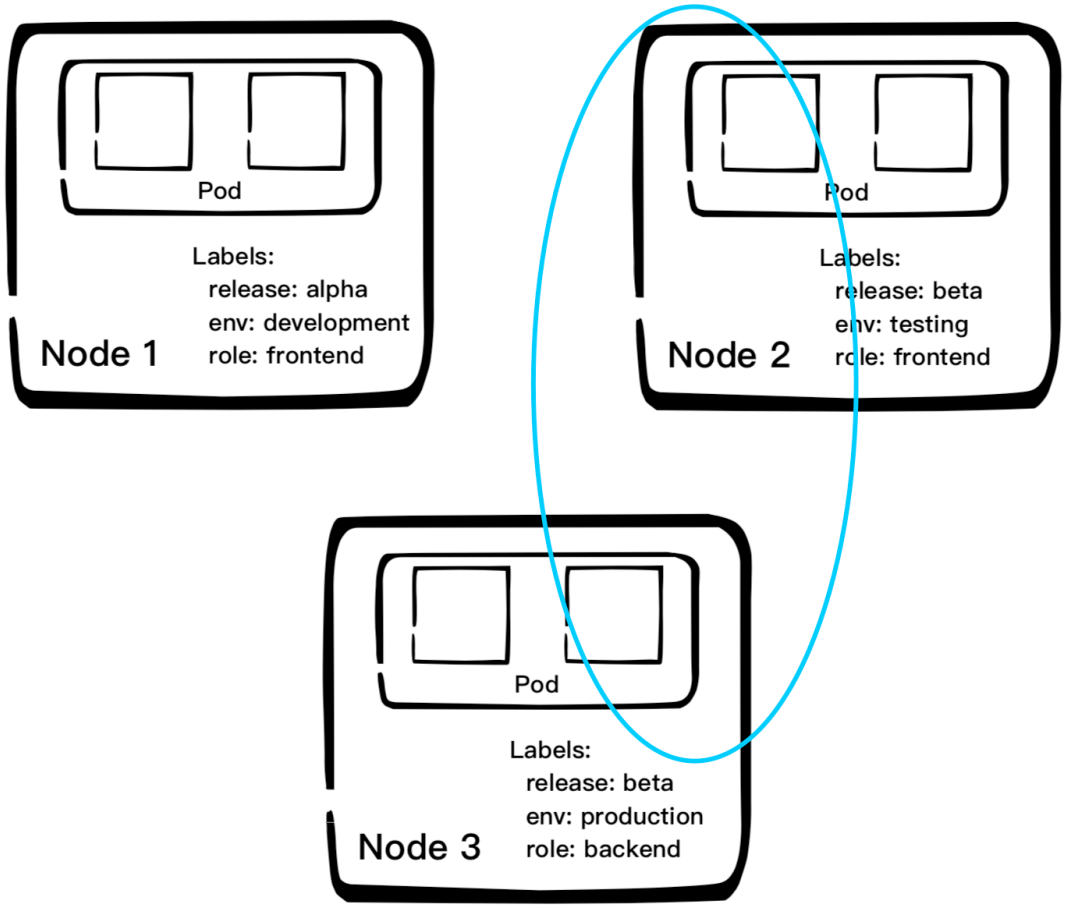
Label Selector 在 Kubernetes 中的重要事宜场景有以下几处。

- kube-controller 进程通过资源对象 RC 上定义的 Label Selector 来筛选要监控的 Pod 副本的数量，从而实现 Pod 副本的数量始终符合预期设定的全自动控制流程。
- kube-proxy 进程通过 Service 的 Label Selector 来选择对应的 Pod，自动建立起每个 Service 到对应 Pod 的请求转发路由表，从而实现 Service 的只能负载均衡机制。
- 通过对某些 Node 定义特定的 Label，并且在 Pod 定义文件中使用 NodeSelector 这种标签调度策略，kube-scheduler 进程可以实现 Pod “定向调度”的特性。

现在来看一个更复杂的例子。假设 Pod 定义了 3 个 Label: release、env 和 role，不同的 Pod 定义了不同的 Label 值并分别部署在不同的 Node 上，如果我们设置了 "role=frontend" 的 Label Selector，则会选取到 Node 1 和 Node 2 上的 Pod。



而设置“release=beta”的 Label Selector，则会选取到 Node 2 和 Node 3 上的 Pod，如下图。



使用 Label 可以给对象创建多组标签，Label 和 Label Selector 共同构成了 Kubernetes 系统中最核心的应用模型，使得被管理对象能够被精细地分组管理，同时实现了整个集群的高可用性。

Annotation

Annotation 与 Label 类似，也使用 key/value 键值对的形式进行定义。不同的是 Label 具有严格的命名规则，它定义的是 Kubernetes 对象的元数据（Metadata），并且用于 Label Selector。而 Annotation 则是用户定义的“附件”信息，以便于外部工具进行查找，很多时候，Kubernetes 的模块自身会通过 Annotation 的方式标记资源对象的一些特殊信息。

通常来说，用 Annotation 来记录的信息如下。

- build 信息、release 信息、Docker 镜像信息等，例如时间戳、release id 号、PR 号、镜像 hash 值、docker registry 地址等。
- 日志库、监控库、分析库等资源库的地址信息。
- 程序调试工具信息，例如工具名称、版本号等。
- 团队的联系信息，例如电话号码、负责人名称、网址等