阿里云 开发者社区

深入浅出 Kubernetes

一次搞懂6个核心原理吃透基础理论

一次学会6个典型问题的华丽操作

作者：声东

Alibaba Group
阿里巴巴集团

阿里云 开发者社区

扫一扫二维码图案，关注我吧

下载更多免费电子书

云服务技术课堂技术圈

云服务技术大学钉钉群

扫码关注阿里巴巴云原生

目录

理论篇

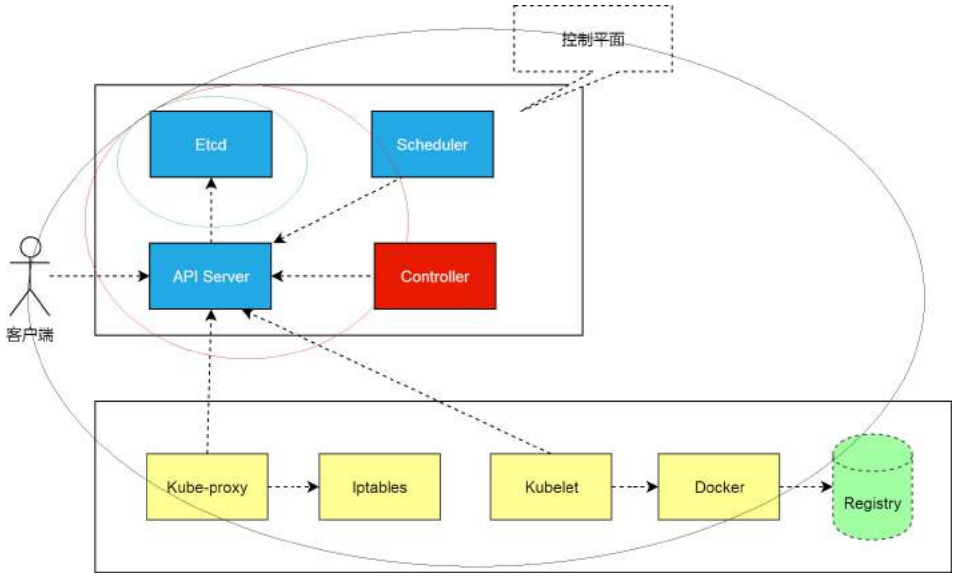
这么理解集群控制器，能行！

简介：当我们尝试去理解 K8S 集群工作原理的时候，控制器肯定是一个难点。这是因为控制器有很多，具体实现大相径庭；且控制器的实现用到了一些较为晦涩的机制，不易理解。但是，我们又不能绕过控制器，因为它是集群的“大脑”。

当我们尝试去理解 K8S 集群工作原理的时候，控制器肯定是一个难点。这是因为控制器有很多，具体实现大相径庭；且控制器的实现用到了一些较为晦涩的机制，不易理解。但是，我们又不能绕过控制器，因为它是集群的“大脑”。今天这篇文章，我们通过分析一个简易冰箱的设计过程，来深入理解集群控制器的产生，功能以及实现方法。

大图

下图是 K8S 集群的核心组件，包括数据库 etcd，调度器 scheduler，集群入口 API Server，控制器 Controller，服务代理 kube-proxy 以及直接管理具体业务容器的 kubelet。这些组件逻辑上可以被分为三个部分：核心组件 etc 数据库，对 etcd 进行直接操作的入口组件 API Server，以及其他组件。这里的“其他组件”之所以可以被划分为一类，是因为它们都可以被看做是集群的控制器。



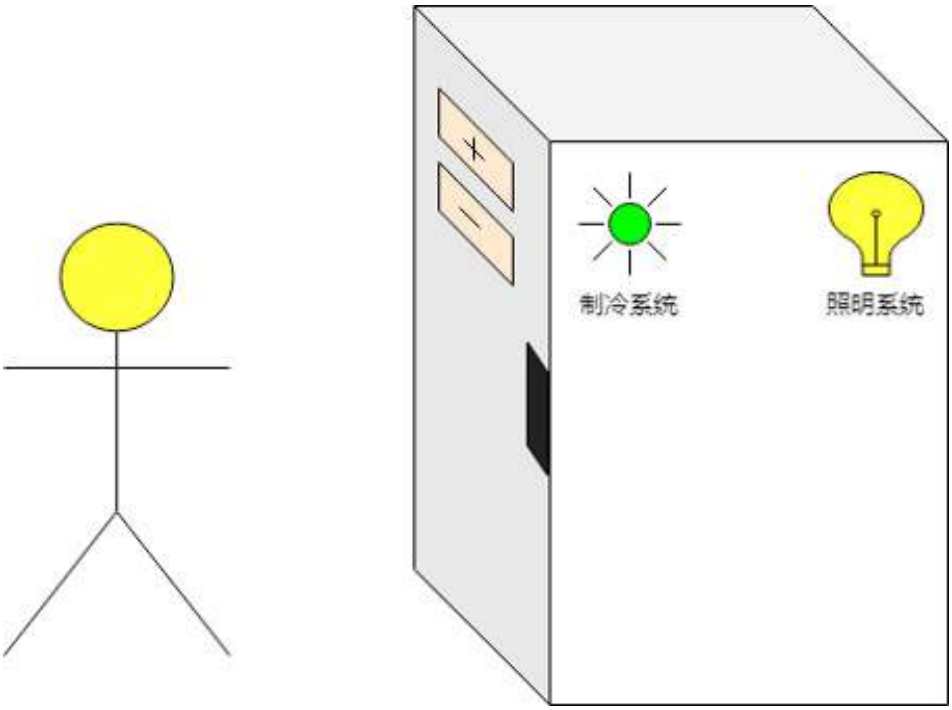
今天我们要讲的就是集群控制器原理。

控制器原理

虽然控制器是 K8S 集群中比较复杂的组件，但控制器本身对我们来说并不陌生的。我们每天使用的洗衣机、冰箱、空调等，都是依靠控制器才能正常工作。在控制器原理这一节，我们通过思考一个简易冰箱的设计过程，来理解 K8S 集群控制器的原理。

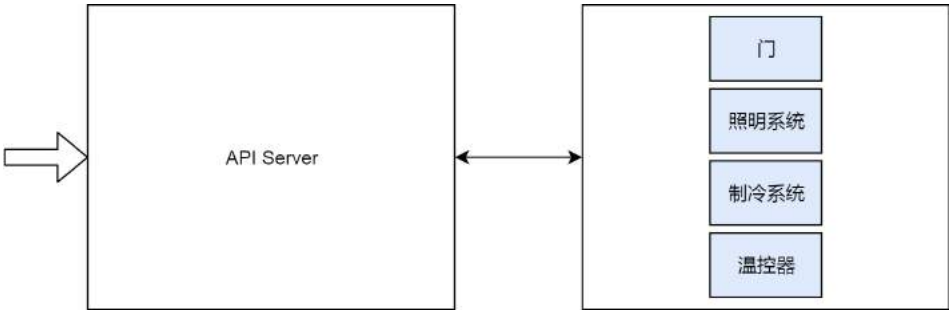
简易的冰箱

这个冰箱包括五个组件：箱体、制冷系统、照明系统、温控器以及门。冰箱只有两个功能：当有人打开冰箱门的时候，冰箱内的灯会自动开启；当有人按下温控器的时候，制冷系统会根据温度设置，调节冰箱内温度。



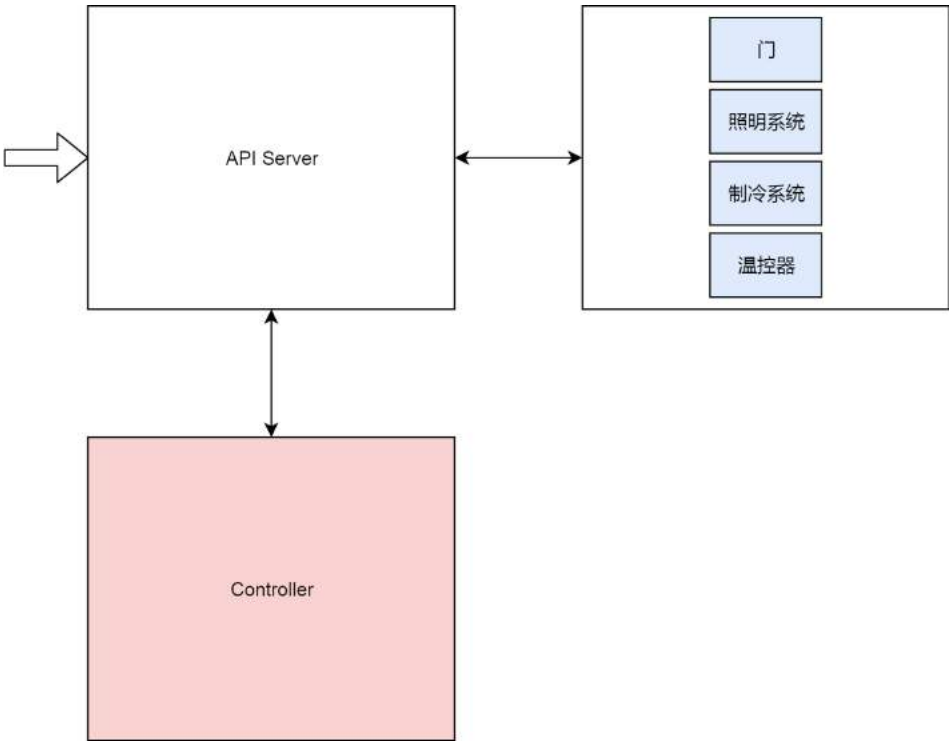
统一入口

对于上边的冰箱，我们可以简单抽象成两个部分：统一的操作入口和冰箱的所有组件。在这里，用户只有通过入口，才能操作冰箱。这个入口提供给用户两个接口：开关门和调节温控器。用户执行这两个接口的时候，入口会分别调整冰箱门和温控器的状态。



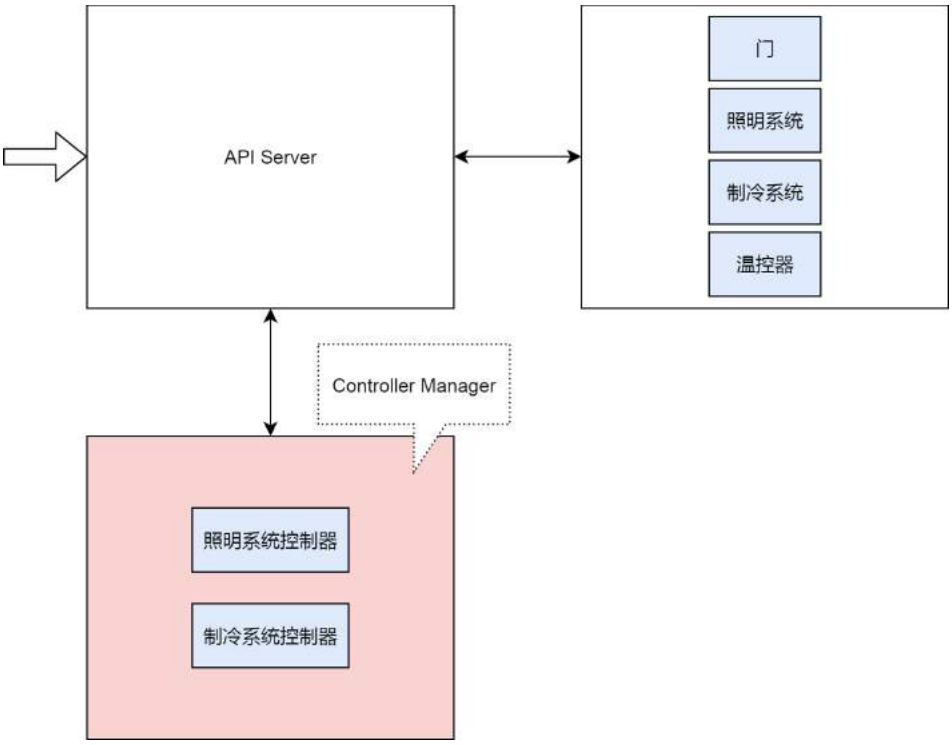
控制器

控制器就是为了解决上边的问题产生的。控制器就是用户的操作，和冰箱各个组件的正确状态之间的一座桥梁：当用户打开门的时候，控制器观察到了门的变化，它替用户打开冰箱内的灯；当用户按下温控器的时候，控制器观察到了用户设置的温度，它替用户管理制冷系统，调节冰箱内温度。



控制器管理器

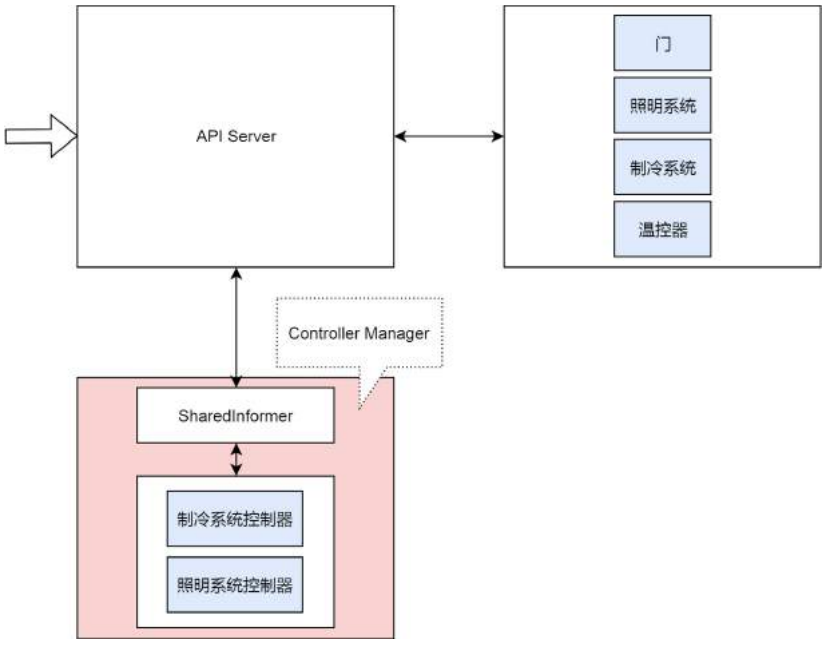
冰箱有照明系统和制冷系统，显然相比一个控制器管理着两个组件，我们替每个组件分别实现一个控制器是更为合理的选择。同时我们实现一个控制器管理器来统一维护所有这些控制器，来保证这些控制器在正常工作。



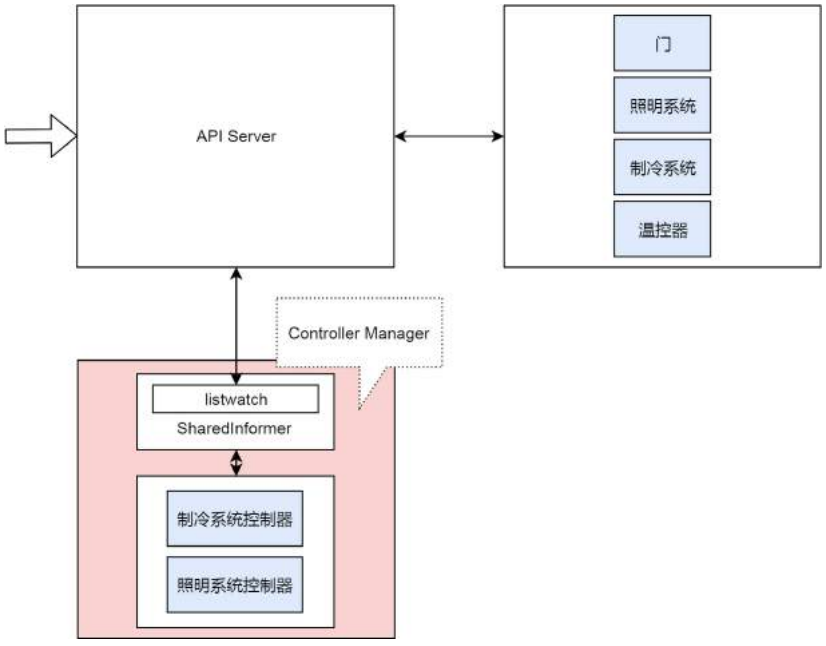
SharedInformer

上边的控制器和控制器管理器，看起来已经相当不错了。但是当冰箱功能增加，势必有很多新的控制器加进来。这些控制器都需要通过冰箱入口，时刻监控自己关心的组件的状态变化。比如照明系统控制器就需要时刻监控冰箱门的状态。当大量控制器不断的和入口通信的时候，就会增加入口的压力。

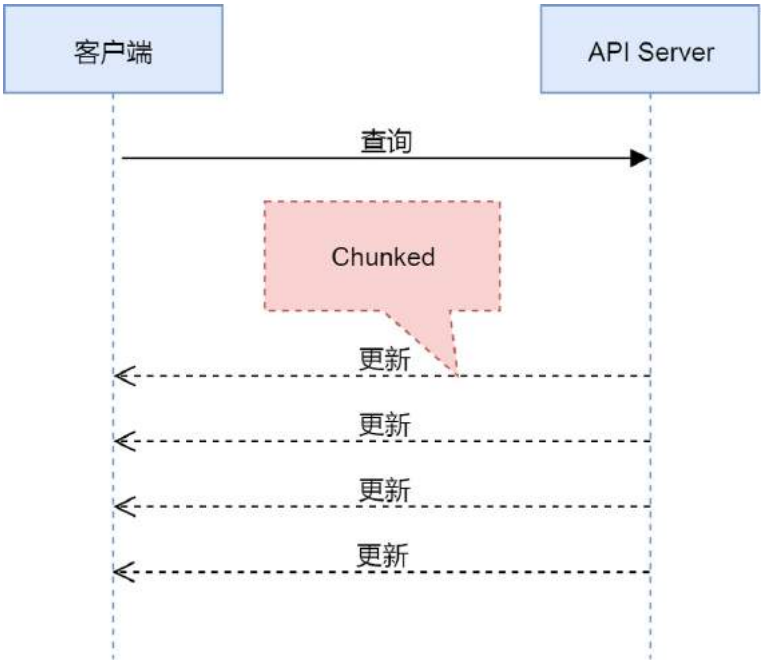
这个时候，我们把监控冰箱组件状态变化这件事情，交给一个新的模块 SharedInformer 来实现。SharedInformer 作为控制器的代理，替控制器监控冰箱组件的状态变化，并根据控制器的喜好，把不同组件状态的变化，通知给对应的控制器。通过优化，这样的 SharedInformer 可以极大的缓解冰箱入口的压力。



ListWatcher



假设 SharedInformer 和冰箱入口通过 http 协议通信的话，那么 http 分块编码（chunked transfer encoding）就是实现 ListWatcher 的一个好的选择。控制器通过 ListWatcher 给冰箱入口发送一个查询然后等待，当冰箱组件有变化的时候，入口通过分块的 http 响应通知控制器。控制器看到 chunked 响应，会认为响应数据还没有发送完成，所以会持续等待。



举例

以上我们从一个简易冰箱的进化过程中，了解了控制器产生的意义，扮演的角色，以及实现的方式。现在我们回到 K8S 集群。K8S 集群实现了大量的控制器，而且在可以预见的未来，新的功能的控制器会不断出现，而一些旧的控制器也会被逐渐淘汰。

目前来说，我们比较常用的控制器，如 pod 控制器、deployment 控制器、service 控制器、replicaset 控制器等。这些控制器一部分是由 kube controller manager 这个管理器实现和管理，而像 route 控制器和 service 控制器，则由

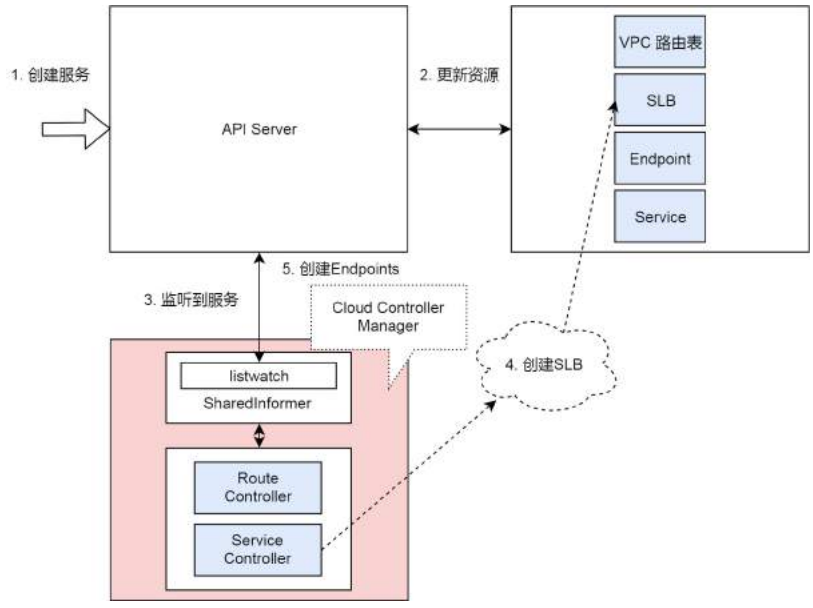
cloud controller manager 实现。

之所以会出现 cloud controller manager，是因为在不同的云环境中，一部分控制器的实现，会因为云厂商、云环境的不同，出现很大的差别。这类控制器被划分出来，由云厂商各自基于 cloud controller manager 分别实现。

这里我们以阿里云 K8S 集群 cloud controller manager 实现的 route 控制器和 service 控制器为例，简单说明 K8S 控制器的工作原理。

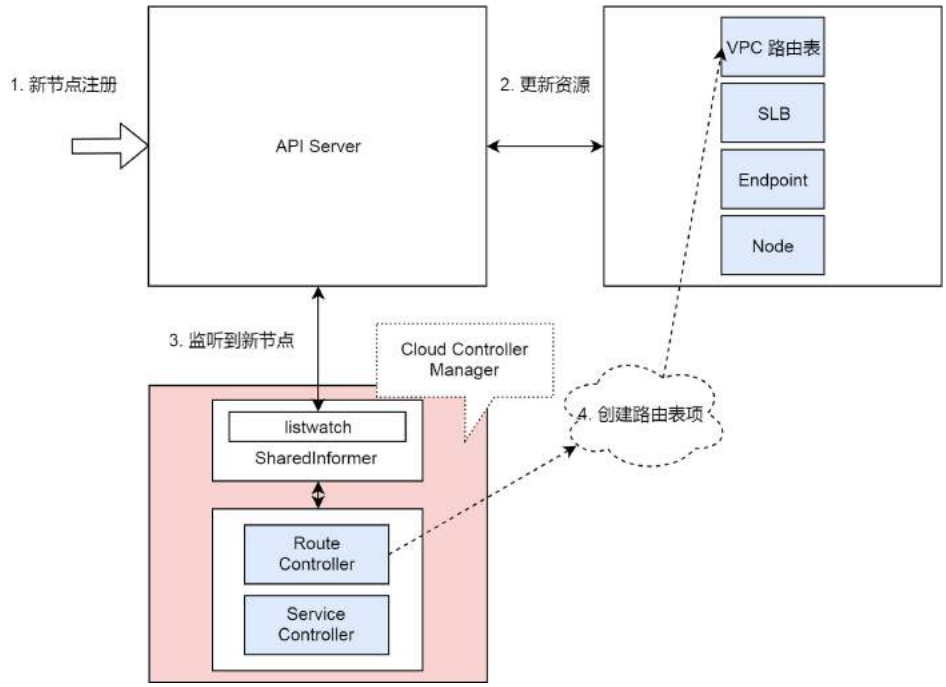
服务控制器

首先，用户请求 API Server 创建一个 LoadBalancer 类型的服务，API Server 收到请求并把这个服务的详细信息写入 etcd 数据库。而这个变化，被服务控制器观察到了。服务控制器理解 LoadBalancer 类型的服务，除了包括存放在 etcd 内部的服务记录之外，还需要一个 SLB 作为服务入口，以及若干 endpoints 作为服务后端。所以服务控制器分别请求 SLB 的云 openapi 和 API Server，来创建云上 SLB 资源，和集群内 endpoints 资源。



路由控制器

在集群网络一章中，我们提到过，当一个节点加入一个 K8S 集群的时候，集群需要在 VPC 路由表里增加一条路由，来搭建这个新加入节点到 pod 网络的主干道。而这件事情，就是路由控制器来做的。路由控制器完成这件事情的流程，与上边服务控制器的处理流程非常类似，这里不再赘述。



结束语

基本上来说，K8S 集群的控制器，其实扮演着集群大脑的角色。有了控制器，K8S 集群才有机会摆脱机械和被动，变成一个自动、智能、有大用的系统。

集群网络详解

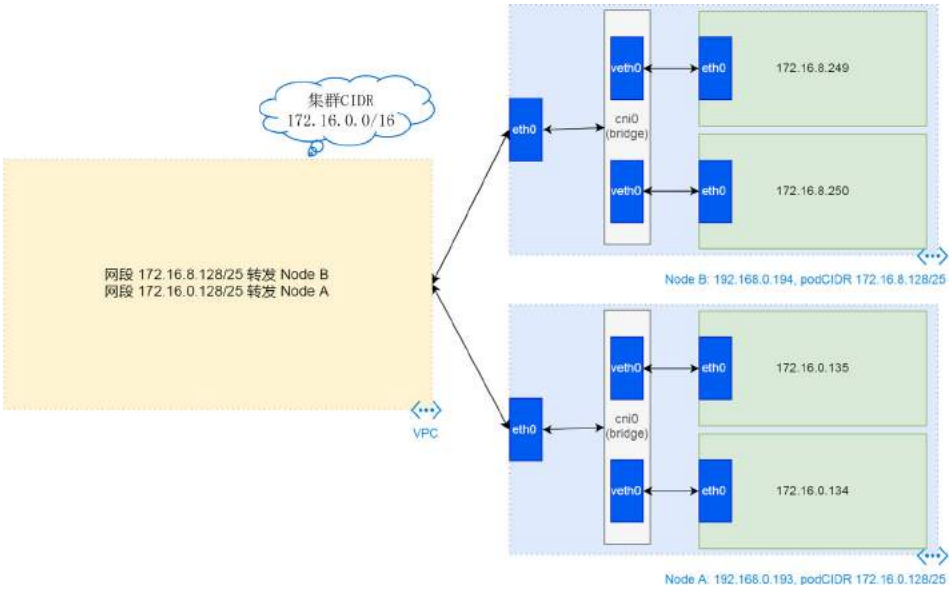
简介：阿里云 K8S 集群网络目前有两种方案，一种是 flannel 方案，另外一种是基于 calico 和弹性网卡 eni 的 terway 方案。Terway 和 flannel 类似，不同的地方在于，terway 支持 Pod 弹性网卡，以及 NetworkPolicy 功能。

阿里云 K8S 集群网络目前有两种方案，一种是 flannel 方案，另外一种是基于 calico 和弹性网卡 eni 的 terway 方案。Terway 和 flannel 类似，不同的地方在于，terway 支持 Pod 弹性网卡，以及 NetworkPolicy 功能。

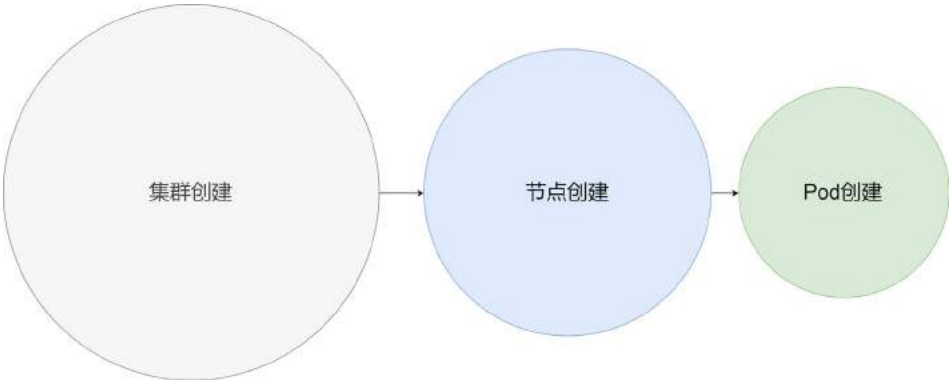
今天这篇文章，我们以 flannel 为例，深入分析阿里云 K8S 集群网络的实现方法。我会从两个角度去分析，一个是网络的搭建过程，另外一个是基于网络的通信。我们的讨论基于当前的 1.12.6 版本。

鸟瞰

总体上来说，阿里云 K8S 集群网络配置完成之后，如下图，包括集群 CIDR，VPC 路由表，节点网络，节点的 podCIDR，节点上的虚拟网桥 cni0，连接 Pod 和网桥的 veth 等部分。



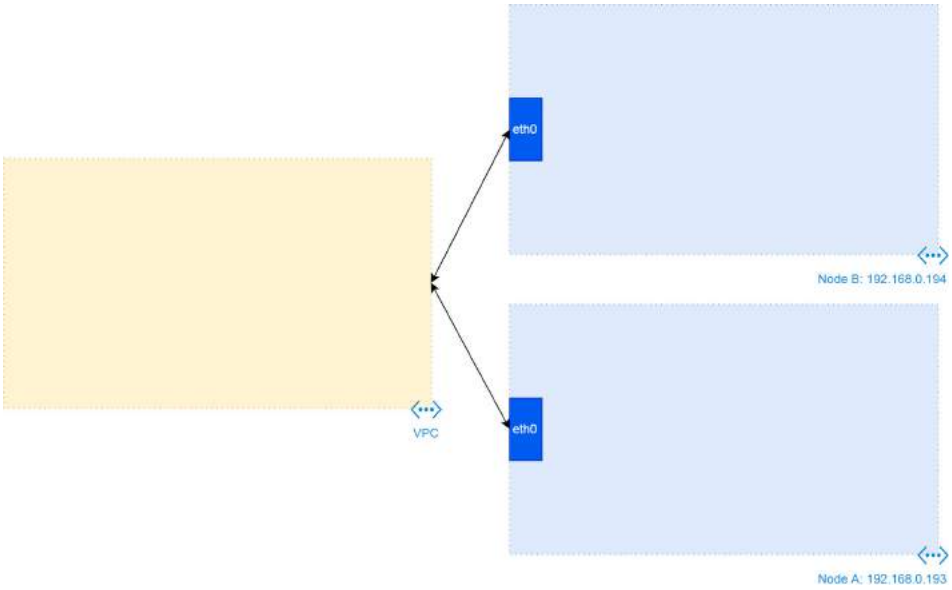
类似的图，大家可能在很多文章中都看过，但是因为其中相关配置过于复杂，比较难理解。这里我们可以把这些配置，分三种情况来理解：集群配置，节点配置以及 Pod 配置。与这三种情况对应的，其实是对集群网络 IP 段的三次划分：首先是集群 CIDR，接着为每个节点分配 podCIDR（即集群 CIDR 的子网段），最后在 podCIDR 里为每个 Pod 分配自己的 IP。



集群网络搭建

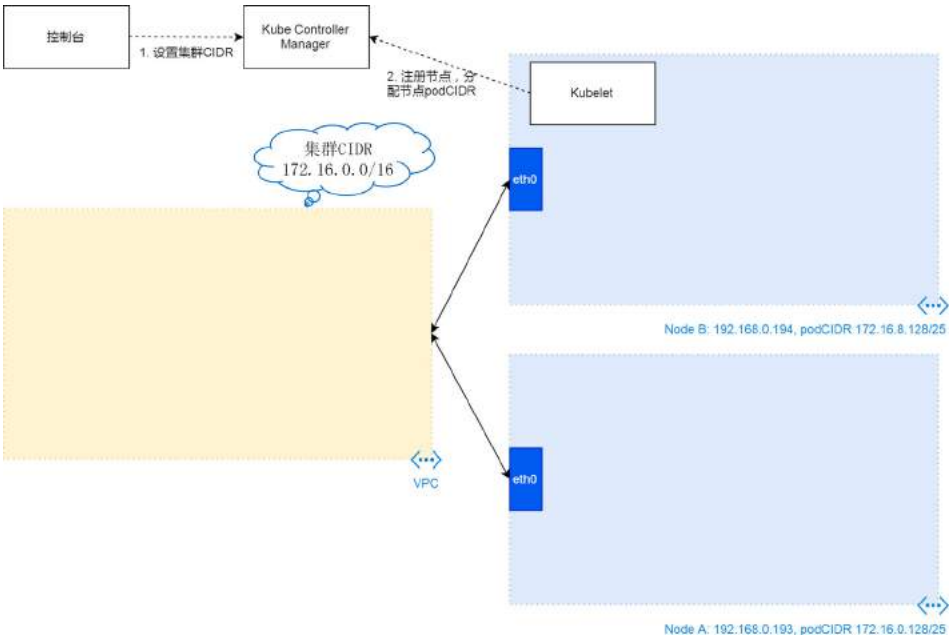
初始阶段

集群的创建，基于云资源 VPC 和 ECS，在创建完 VPC 和 ECS 之后，我们基本上可以得到如下图的资源配置。我们得到一个 VPC，这个 VPC 的网段是 192.168.0.0/16，我们得到若干 ECS，他们从 VPC 网段里分配到 IP 地址。



集群阶段

在以上出初始资源的基础上，我们利用集群创建控制台得到集群 CIDR。这个值会以参数的形式传给集群节点 provision 脚本，并被脚本传给集群节点配置工具 kubeadm。kubeadm 最后把这个参数写入集群控制器静态 Pod 的 yaml 文件 kube-controller-manager.yaml。

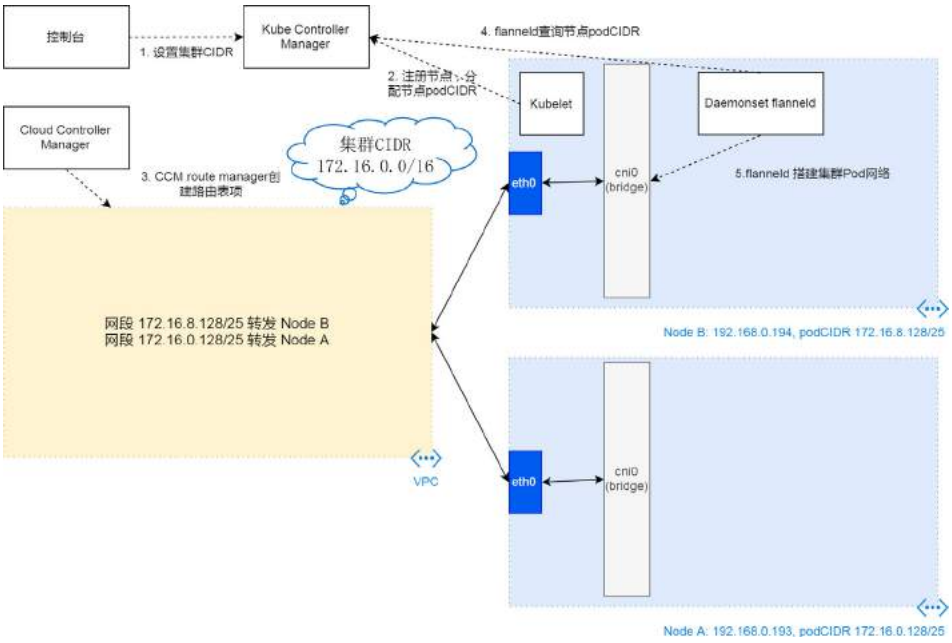


集群控制器有了这个参数，在节点 kubelet 注册节点到集群的时候，集群控制器会为每个注册节点，划分一个子网出来，即为每个节点分配 podCIDR。如上图，Node B 的子网是 172.16.8.1/25，而 Node A 的子网是 172.16.0.128/25。这个配置会记录到集群 node 的 podCIDR 数据项里。

节点阶段

经过以上集群阶段，K8S 有了集群 CIDR，以及为每个节点划分的 podCIDR。在此基础上，集群会下发 flanneld 到每个阶段上，进一步搭建节点上，可以给 Pod 使用的网络框架。这里主要有两个操作，第一个是集群通过 Cloud Controller Manager 给 VPC 配置路由表项。路由表项对每个节点有一条。每一条的意思是，如果 VPC 路由收到目的地址是某一个节点 podCIDR 的 IP 地址，那么路由会把这个网络包转发到对应的 ECS 上。第二个是创建虚拟网桥 cni0，以及与 cni0 相关的路由。这些配置的作用是，从阶段外部进来的网络包，如果目的 IP 是 podCIDR，则会被节点转发到 cni0 虚拟局域网里。

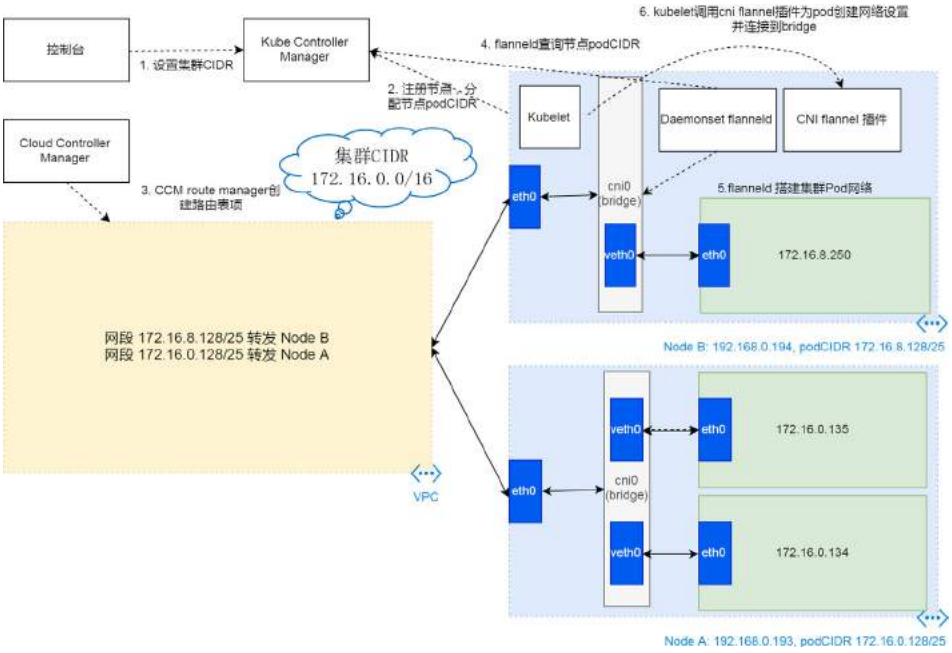
注意：实际实现上，cni0 的创建，是在第一个使用 Pod 网络的 Pod 被调度到节点上的时候，由下一节中 flannal cni 创建的，但是从逻辑上来说，cni0 属于节点网络，不属于 Pod 网络，所以在此描述。



Pod 阶段

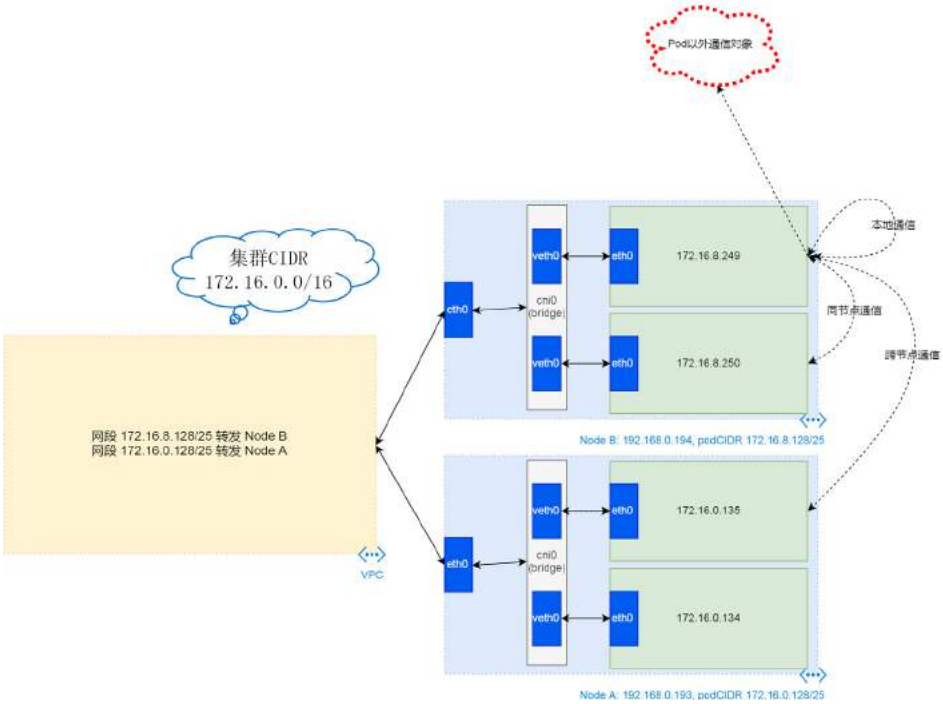
在前边的三个阶段，集群实际上已经为 Pod 之间搭建了网络通信的干道。这个时候，如果集群把一个 Pod 调度到节点上，kubelet 会通过 flannel cni 为这个 Pod 本身创建网络命名空间和 veth 设备，然后，把其中一个 veth 设备加入到 cni0 虚拟网桥里，并为 Pod 内的 veth 设备配置 ip 地址。这样 Pod 就和网络通信的干道连接在了一起。这里需要强调的是，前一节的 flanneld 和这一节的 flannel cni 完全是两个组件。flanneld 是一个 daemonset 下发到每个节点的 pod，它的作用是搭建网络（干道），而 flannel cni 是节点创建的时候，通过 kubernetes-cni 这个 rpm 包安装的 cni 插件，其被 kubelet 调用，用来为具体的 pod 创建网络（分枝）。

理解这两者的区别，有助于我们理解 flanneld 和 flannel cni 相关的配置文件的用途。比如 /run/flannel/subnet.env，是 flanneld 创建的，为 flannel cni 提供输入的一个环境变量文件；又比如 /etc/cni/net.d/10-flannel.conf，也是 flanneld pod（准确的说，是 pod 里的脚本 install-cni）从 pod 里拷贝到节点目录，给 flannel cni 使用的子网配置文件。



通信

以上完成 Pod 网络环境搭建。基于以上的网络环境，Pod 可以完成四种通信：本地通信，同节点 Pod 通信，跨节点 Pod 通信，以及 Pod 和 Pod 网络之外的实体通信。



其中本地通信，说的是 Pod 内部，不同容器之前通信。因为 Pod 内网容器之间共享一个网络协议栈，所以他们之间的通信，可以通过 loopback 设备完成。

同节点 Pod 之间的通信，是 cni0 虚拟网桥内部的通信，这相当于一个二层局域网内部设备通信。

跨节点 Pod 通信略微复杂一点，但也很直观，发送端数据包，通过 cni0 网桥的网关，流转到节点上，然后经过节点 eth0 发送给 VPC 路由。这里不会经过任何封包操作。当 VPC 路由收到数据包时，它通过查询路由表，确认数据包目的地，并把数据包发送给对应的 ECS 节点。而进去节点之后，因为 flanneld 在节点上创建了真的 cni0 的路由，所以数据包会被发送到目的地的 cni0 局域网，再到目的地 Pod。

最后一种情况，Pod 与非 Pod 网络的实体通信，需要经过节点上 iptables 规则做 snat，而此规则就是 flanneld 依据命令行 --ip-masq 选项做的配置。

总结

以上是阿里云 K8S 集群网络的搭建和通信原理。我们主要通过网络搭建和通信两个角度去分析 K8S 集群网络。其中网络搭建包括初始阶段，集群阶段，节点阶段以及 Pod 阶段，这么分类有助于我们理解这些复杂的配置。而理解了各个配置，集群通信原理就比较容易理解了。

集群伸缩原理

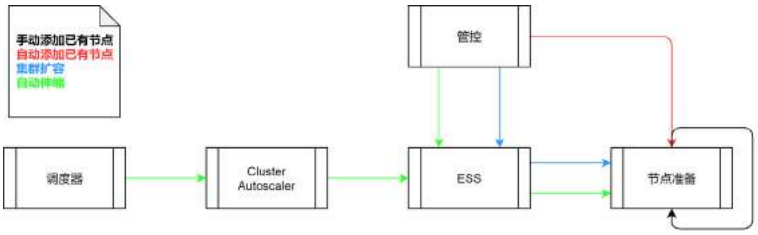
简介：阿里云 K8S 集群的一个重要特性，是集群的节点可以动态的增加或减少。有了这个特性，集群才能在计算资源不足的情况下扩容新的节点，同时也可以在资源利用率降低的时候，释放节点以节省费用。 这篇文章，我们讨论阿里云 K8S 集群扩容与缩容的实现原理。

阿里云 K8S 集群的一个重要特性，是集群的节点可以动态的增加或减少。有了这个特性，集群才能在计算资源不足的情况下扩容新的节点，同时也可以在资源利用率降低的时候，释放节点以节省费用。

这篇文章，我们讨论阿里云 K8S 集群扩容与缩容的实现原理。理解实现原理，在遇到问题的时候，我们就可以高效地排查并定位原因。我们的讨论基于当前的 1.12.6 版本。

节点增加原理

阿里云 K8S 集群可以给集群增加节点的方式有，添加已有节点，集群扩容，和自动伸缩。其中，添加已有节点又可分为手动添加已有节点和自动添加已有节点。节点的增加涉及到的组件有，节点准备，弹性伸缩（ESS），管控，Cluster Autoscaler 以及调度器。



手动添加已有节点

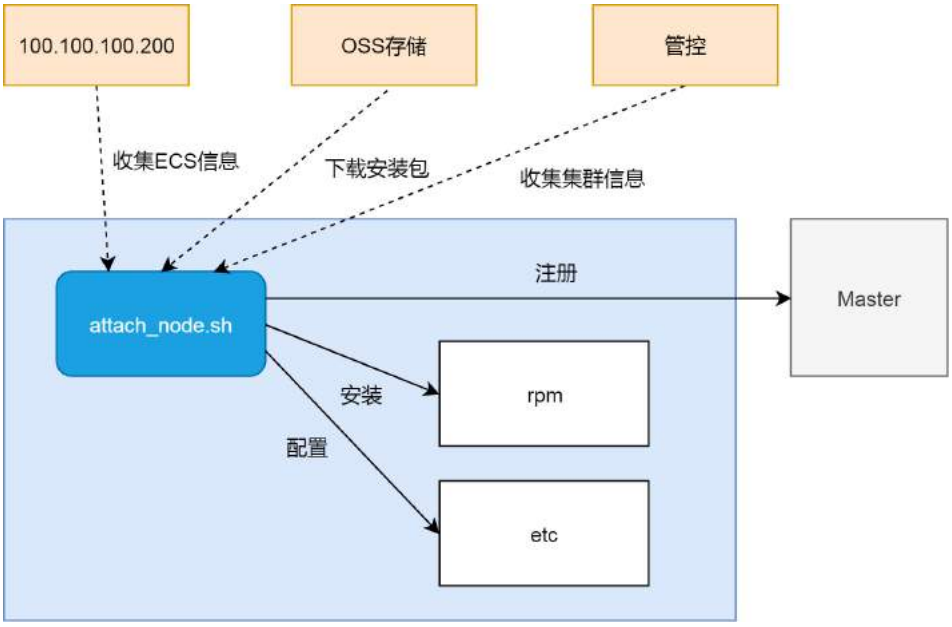
节点准备，其实就是把一个普通的 ECS 实例，安装配置成为一个 K8S 集群节点的过程。这个过程仅靠一条命令就可以完成。这条命令使用 curl 下载 attach_node.sh 脚本，然后以 openapi token 为参数，在 ECS 上运行。

curl http:///public/pkg/run/attach//attach_node.sh | bash -s -- --openapi-token

这里 token 是一个对的 key，而 value 是当前集群的基本信息。阿里云 K8S 集群的管控，在接到手动添加已有节点请求的时候，会生成这个对，并把 key 作为 token 返回给用户。

这个 token（key）存在的价值，是其可以让 attach_node.sh 脚本，以匿名身份在 ECS 上索引到集群的基本信息（value），而这些基本信息，对节点准备至关重要。

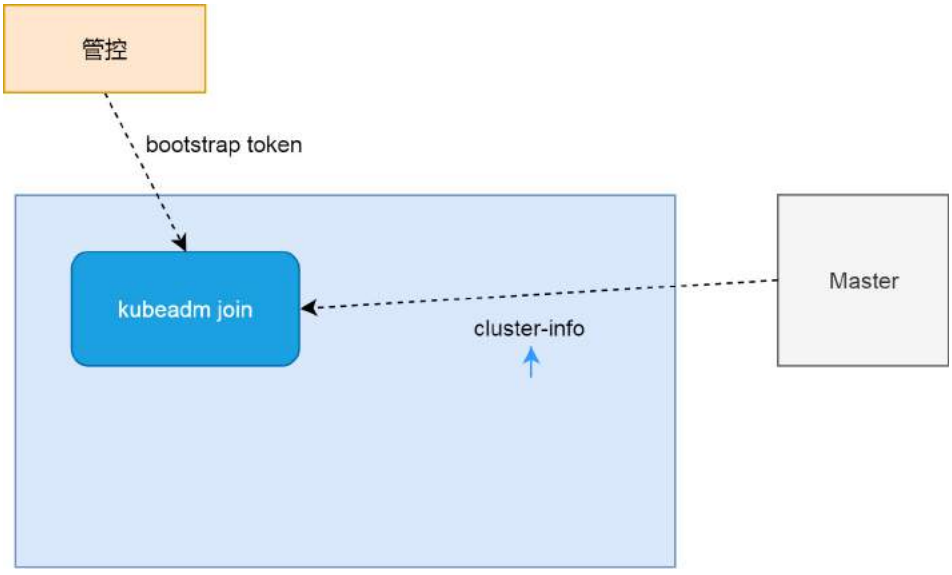
总体上来说，节点准备就做两件事情，读和写。读即数据收集，写即节点配置。



这里的读写过程，绝大部分都很基础，大家可以通过阅读脚本来了解细节。唯一需要特别说明的是，kubeadm join 把节点注册到 Master 的过程。此过程需要新加节点和集群 Master 之间建立互信。

一边，新加节点从管控处获取的 bootstrap token（与 openapi token 不同，此 token 是 value 的一部分内容），实际上是管控通过可信的途径从集群 Master 上获取的。新加节点使用这个 bootstrap token 连接 Master，Master 则可通过验证这个 bootstrap token 来建立对新加节点的信任。

另一边，新加节点以匿名身份从 Master kube-public 命名空间中获取集群 cluster-info，cluster-info 包括集群 CA 证书，和使用集群 bootstrap token 对这个 CA 做的签名。新加节点使用从管控处获取的 bootstrap token，对 CA 生成 b 新的签名，然后将此签名与 cluster-info 内签名做对比，如果两个签名一致，则说明 cluster-info 和 bootstrap token 来自同一集群。新加节点因为信任管控，所以建立对 Master 的信任。



自动添加已有节点

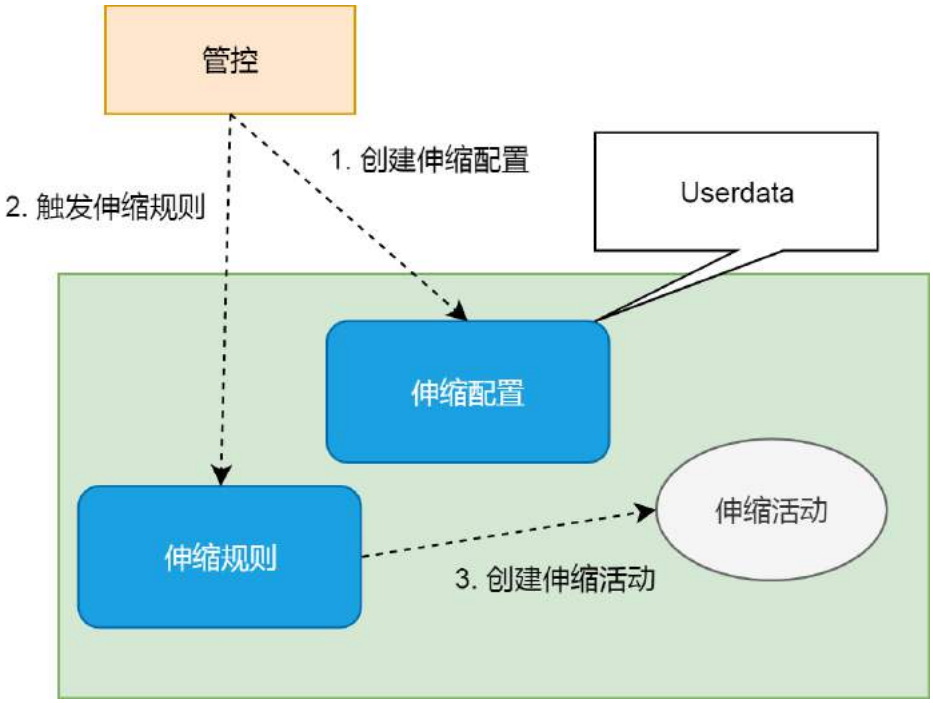
自动添加已有节点，不需要人为拷贝黏贴脚本到 ECS 命令行来完成节点准备的过程。管控使用了 ECS userdata 的特性，把类似以上节点准备的脚本，写入 ECS userdata，然后重启 ECS 并更换系统盘。当 ECS 重启之后，会自动执行 Userdata 里边的脚本，来完成节点添加的过程。这部分内容，大家其实可以通过查看节点 userdata 来确认。

```
!/bin/bash

mkdir -p /var/log/acs

curl http:///public/pkg/run/attach/1.12.6-aliyun.1/attach_node.sh | bash -s -- --docker-version --token --endpoint --cluster-dns > /var/log/acs/init.log
```

这里我们看到，attach_node.sh 的参数，与前一节的参数有很大的不同。其实这里的参数，都是前一节 value 的内容，即管控创建并维护的集群基本信息。自动添加已有节点省略了通过 key 获取 value 的过程。

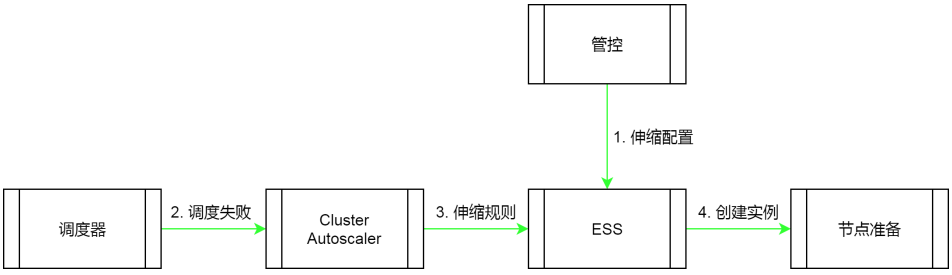
集群扩容

集群扩容与以上添加已有节点不同，此功能针对需要新购节点的情形。集群扩容的实现，在添加已有节点的基础上，引入了弹性伸缩 ESS 组件。ESS 组件负责从无到有的过程，而剩下的过程与添加已有节点类似，即依靠 ECS userdata 脚本来完成节点准备。下图是管控通过 ESS 从无到有创建 ECS 的过程。



自动伸缩

前边三种方式是需要人为干预的伸缩方式，而自动伸缩的本质不同，是它可以在业务需求量增加的时候，自动创建 ECS 实例并加入集群。为了实现自动化，这里引入了另外一个组件 Cluster Autoscaler。集群自动伸缩包括两个独立的过程。



其中第一个过程，主要用来配置节点的规格属性，包括设置节点的用户数据。这

个用户数据和手动添加已有节点的脚本类似，不同的地方在于，其针对自动伸缩这种场景，增加了一些专门的标记。attach_node.sh 脚本会根据这些标记，来设置节点的属性。

```
!/bin/sh

curl http:///public/pkg/run/attach/1.12.6-aliyun.1/attach_node.sh | bash -s -- --openapi-token --ess true --labels k8s.io/cluster-auto-scaler=true,workload_type=cpu,k8s.aliyun.com=true
```

而第二个过程，是实现自动增加节点的关键。这里引入了一个新的组件 Autoscaler，它以 Pod 的形式运行在 K8S 集群中。理论上来说，我们可以把这个组件当做一个控制器。因为它的作用与控制器类似，基本上还是监听 Pod 状态，以便在 Pod 因为节点资源不足而不能被调度的时，去修改 ESS 的伸缩规则来增加新的节点。

这里有一个知识点，集群调度器衡量资源是否充足的标准，是“预订率”，而不是“使用率”。这两者的差别，类似酒店房价预订率和实际入住率：完全有可能有人预订了酒店，但是并没有实际入住。在开启自动伸缩功能的时候，我们需要设置缩容阈值，就是“预订率”的下线。之所以不需要设置扩容阈值。是因为 Autoscaler 扩容集群，依靠的是 Pod 的调度状态：当 Pod 因为节点资源“预订率”太高无法被调度的时候，Autoscaler 就会扩容集群。

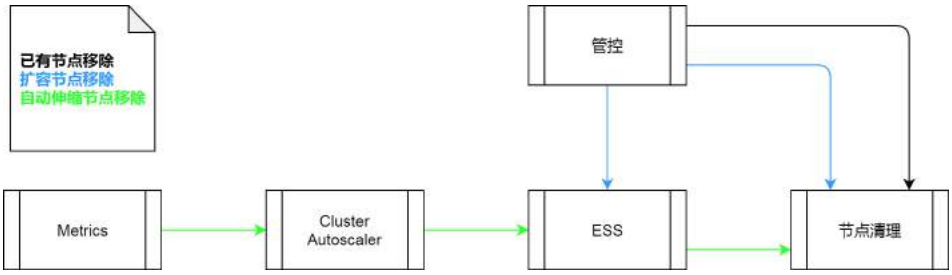
节点减少原理

与增加节点不同，集群减少节点的操作只有一个移除节点的入口。但对于用不同方法加入的节点，其各自移除方式略有不同。

首先，通过添加已有节点加入的节点，需要三步去移除：管控通过 ECS API 清楚 ECS userdata；管控通过 K8S API 从集群中删除节点；管控通过 ECS Invoke-

Command 在 ECS 上执行 kubeadm reset 命令清理节点。

其次，通过集群扩容加入的节点，则在上边的基础上，增加了断开 ESS 和 ECS 关系的操作。此操作由管控调用 ESS API 完成。



最后，经过 Cluster Autoscaler 动态增加的节点，则在集群 CPU 资源“预订率”降低的时候，由 Cluster Autoscaler 自动移除释放。其触发点是 CPU“预订率”，即上图写 Metrics 的原因。

总结

总体上来说，K8S 集群节点的增加与减少，主要涉及四个组件，分别是 Cluster Autoscaler，ESS，管控以及节点本身（准备或清理）。根据场景不同，我们需要排查不同的组件。其中 Cluster Autoscaler 是一个普通的 Pod，其日志的获取和其他 Pod 无异；ESS 弹性伸缩有其专门的控制台，我们可以在控制台排查其伸缩配置、伸缩规则等相关子实例日志和状态；而管控的日志，可以通过查看日志功能来查看；最后，对于节点的准备与清理，其实就是排查对应的脚本的执行过程。

以上讲道理居多，希望对大家排查问题有所帮助。

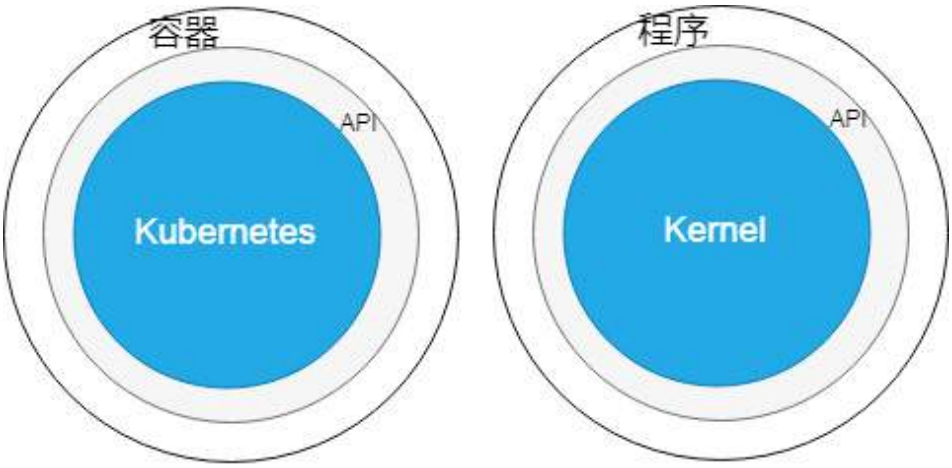
认证与调度

简介：不知道大家有没有意识到一个现实，就是大部分时候，我们已经不像以前一样，通过命令行，或者可视窗口来使用一个系统了。现在我们上微博、或者网购，操作的其实不是眼前这台设备，而是一个又一个集群。 通常，这样的集群拥有成百上千个节点，每个节点是一台物理机或虚拟机。

不知道大家有没有意识到一个现实，就是大部分时候，我们已经不像以前一样，通过命令行，或者可视窗口来使用一个系统了。现在我们上微博、或者网购，操作的其实不是眼前这台设备，而是一个又一个集群。

通常，这样的集群拥有成百上千个节点，每个节点是一台物理机或虚拟机。集群一般远离用户，坐落在数据中心。为了让这些节点互相协作，对外提供一致且高效的服务，集群需要操作系统。Kubernetes 就是这样的操作系统。

比较 Kubernetes 和单机操作系统，Kubernetes 相当于内核，它负责集群软硬件资源管理，并对外提供统一的入口，用户可以通过这个入口来使用集群，和集群沟通。



而运行在集群之上的程序，与普通程序有很大的不同。这样的程序，是“关在笼子里”的程序。它们从被制作，到被部署，再到被使用，都不寻常。我们只有深挖根源，才能理解其本质。

“关在笼子里”的程序

代码

我们使用 go 语言写了一个简单的 web 服务器程序 app.go，这个程序监听在 2580 这个端口。通过 http 协议访问这个服务的根路径，服务会返回“This is a small app for kubernetes...”字符串。

```
package main

import (

        "github.com/gorilla/mux"

        "log"

        "net/http"

)

func about(w http.ResponseWriter, r *http.Request) {
```

```
        w.Write([]byte("This is a small app for kubernetes...\n"))
}
func main() {
        r := mux.NewRouter()
        r.HandleFunc("/", about)
        log.Fatal(http.ListenAndServe("0.0.0.0:2580", r))
}
```

使用 go build 命令编译这个程序，产生 app 可执行文件。这是一个普通的可执行文件，它在操作系统里运行，会依赖系统里的库文件。

```
# ldd app
linux-vdso.so.1 => (0x00007ffd1f7a3000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f554fd4a000)
libc.so.6 => /lib64/libc.so.6 (0x00007f554f97d000)
/lib64/ld-linux-x86-64.so.2 (0x00007f554ff66000)
```

“笼子”

为了让这个程序不依赖于操作系统自身的库文件，我们需要制作容器镜像，即隔离的运行环境。Dockerfile 是制作容器镜像的“菜谱”。我们的菜谱就只有两个步骤，下载一个 centos 的基础镜像，把 app 这个可执行文件放到镜像中 /usr/local/bin 目录中去。

```
FROM centos
ADD app /usr/local/bin
```

地址

制作好的镜像存再本地，我们需要把这个镜像上传到镜像仓库里去。这里的镜像仓库，相当于应用商店。我们使用阿里云的镜像仓库，上传之后镜像地址是：

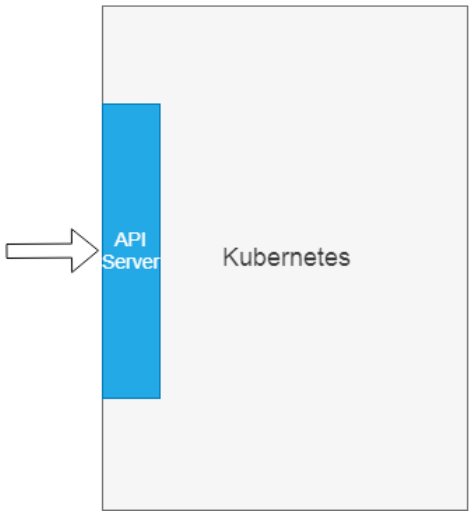
```
registry.cn-hangzhou.aliyuncs.com/kube-easy/app:latest
```

镜像地址可以拆分成四个部分：仓库地址 / 命名空间 / 镜像名称 : 镜像版本。显然，镜像上边的镜像，在阿里云杭州镜像仓库，使用的命名空间是 kube-easy，镜像名 : 版本是 app:latest。至此，我们有了一个可以在 Kubernetes 集群上运行的，“关在笼子里”的小程序。

得其门而入

入口

Kubernetes 作为操作系统，和普通的操作系统一样，有 API 的概念。有了 API，集群就有了入口；有了 API，我们使用集群，才能得其门而入。Kubernetes 的 API 被实现为运行在集群节点上的组件 API Server。这个组件是典型的 web 服务器程序，通过对外暴露 http(s) 接口来提供服务。



这里我们创建一个阿里云 Kubernetes 集群。登录集群管理页面，我们可以看到 API Server 的公网入口。

```
API Server 内网连接端点：https://xx.xxx.xxx.xxx:6443
```

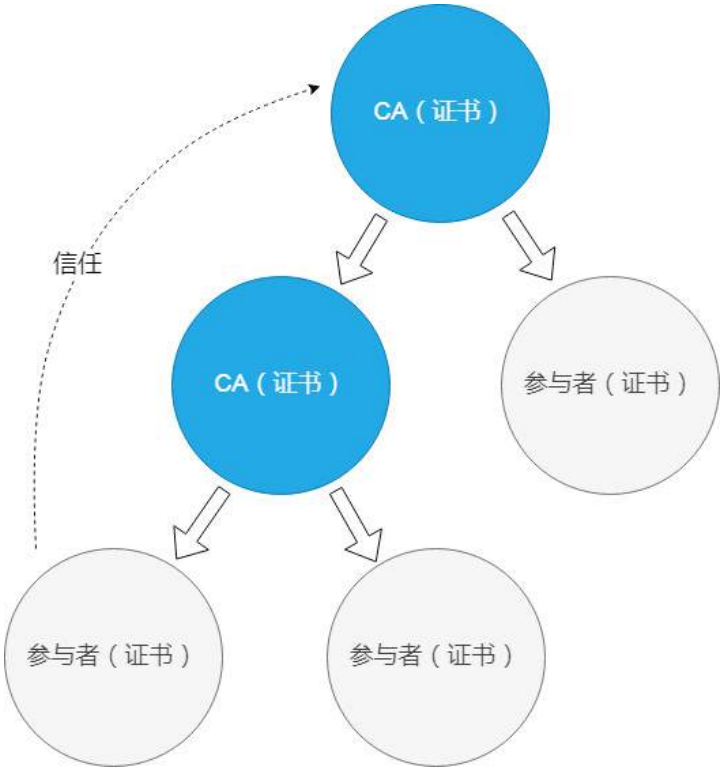
双向数字证书验证

阿里云 Kubernetes 集群 API Server 组件，使用基于 CA 签名的双向数字证书认证来保证客户端与 api server 之间的安全通信。这句话很绕口，对于初学者不太好理解，我们来深入解释一下。

从概念上来讲，数字证书是用来验证网络通信参与者的一个文件。这和学校颁发给学生的毕业证书类似。在学校和学生之间，学校是可信第三方 CA，而学生是通信参与者。如果社会普遍信任一个学校的声誉的话，那么这个学校颁发的毕业证书，也会得到社会认可。参与者证书和 CA 证书可以类比毕业证和学校的办学许可证。

这里我们有两类参与者，CA 和普通参与者；与此对应，我们有两种证书，CA 证书和参与者证书；另外我们还有两种关系，证书签发关系，以及信任关系。这两种关系至关重要。

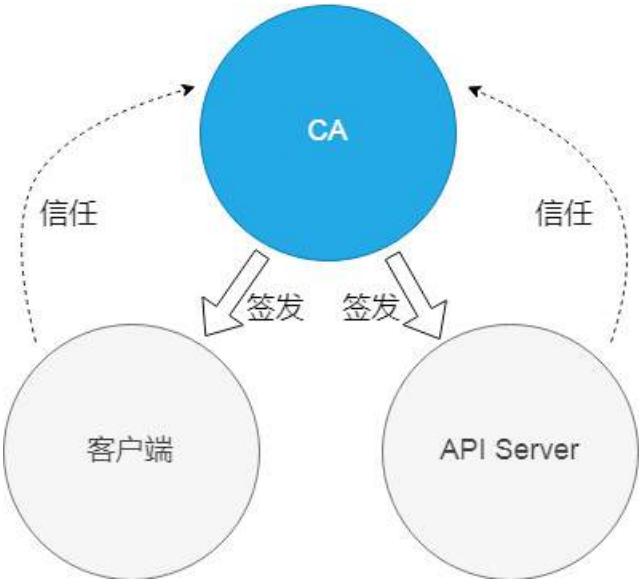
我们先看签发关系。如下图，我们有两张 CA 证书，三个参与者证书。其中最上边的 CA 证书，签发了两张证书，一张是中间的 CA 证书，另一张是右边的参与者证书；中间的 CA 证书，签发了下边两张参与者证书。这六张证书以签发关系为联系，形成了树状的证书签发关系图。



然而，证书以及签发关系本身，并不能保证可信的通信可以在参与者之间进行。以上图为例，假设最右边的参与者是一个网站，最左边的参与者是一个浏览器，浏览器相信网站的数据，不是因为网站有证书，也不是因为网站的证书是 CA 签发的，而是因为浏览器相信最上边的 CA，也就是信任关系。

理解了 CA（证书），参与者（证书），签发关系，以及信任关系之后，我们回过头来看“基于 CA 签名的双向数字证书认证”。客户端和 API Server 作为通信的普通参与者，各有一张证书。而这两张证书，都是由 CA 签发，我们简单称它们为集群 CA 和客户端 CA。客户端信任集群 CA，所以它信任拥有集群 CA 签发证书的 API Server；反过来 API Server 需要信任客户端 CA，它才愿意与客户端通信。

阿里云 Kubernetes 集群，集群 CA 证书，和客户端 CA 证书，实现上其实是一张证书，所以我们有这样的关系图。



KubeConfig 文件

登录集群管理控制台，我们可以拿到 KubeConfig 文件。这个文件包括了客户端证书，集群 CA 证书，以及其他。证书使用 base64 编码，所以我们可以使用 base64 工具解码证书，并使用 openssl 查看证书文本。

- 首先，客户端证书的签发者 CN 是集群 id c0256a3b8e4b948bb9c21e66b0e-1d9a72，而证书本身的 CN 是子账号 252771643302762862。

```
Certificate:

    Data:

        Version: 3 (0x2)

        Serial Number: 787224 (0xc0318)

    Signature Algorithm: sha256WithRSAEncryption

        Issuer: O=c0256a3b8e4b948bb9c21e66b0e1d9a72, OU=default,
CN=c0256a3b8e4b948bb9c21e66b0e1d9a72
```

```
        Validity

            Not Before: Nov 29 06:03:00 2018 GMT

            Not After : Nov 28 06:08:39 2021 GMT

        Subject: O=system:users, OU=, CN=252771643302762862
```

- 其次，只有在 API Server 信任客户端 CA 证书的情况下，上边的客户端证书才能通过 API Server 的验证。kube-apiserver 进程通过 client-ca-file 这个参数指定其信任的客户端 CA 证书，其指定的证书是 /etc/kubernetes/pki/apiserver-ca.crt。这个文件实际上包含了两张客户端 CA 证书，其中一张和集群管控有关系，这里不做解释，另外一张如下，它的 CN 与客户端证书的签发者 CN 一致。

```
Certificate:

    Data:

        Version: 3 (0x2)

        Serial Number: 787224 (0xc0318)

    Signature Algorithm: sha256WithRSAEncryption

        Issuer: O=c0256a3b8e4b948bb9c21e66b0e1d9a72, OU=default,
CN=c0256a3b8e4b948bb9c21e66b0e1d9a72

        Validity

            Not Before: Nov 29 06:03:00 2018 GMT

            Not After : Nov 28 06:08:39 2021 GMT

        Subject: O=system:users, OU=, CN=252771643302762862
```

- 再次，API Server 使用的证书，由 kube-apiserver 的参数 tls-cert-file 决定，这个参数指向证书 /etc/kubernetes/pki/apiserver.crt。这个证书的 CN 是 kube-apiserver，签发者是 c0256a3b8e4b948bb9c21e66b0e-1d9a72，即集群 CA 证书。

```
Certificate:

    Data:

        Version: 3 (0x2)

        Serial Number: 2184578451551960857 (0x1e512e86fcba3f19)

    Signature Algorithm: sha256WithRSAEncryption

        Issuer: O=c0256a3b8e4b948bb9c21e66b0e1d9a72, OU=default,
CN=c0256a3b8e4b948bb9c21e66b0e1d9a72

        Validity

            Not Before: Nov 29 03:59:00 2018 GMT

            Not After : Nov 29 04:14:23 2019 GMT

        Subject: CN=kube-apiserver
```

- 最后，客户端需要验证上边这张 API Server 的证书，因而 KubeConfig 文件里包含了其签发者，即集群 CA 证书。对比集群 CA 证书和客户端 CA 证书，发现两张证书完全一样，这符合我们的预期。

```
Certificate:

    Data:

        Version: 3 (0x2)

        Serial Number: 786974 (0xc021e)

    Signature Algorithm: sha256WithRSAEncryption

        Issuer: C=CN, ST=ZheJiang, L=HangZhou, O=Alibaba, OU=ACS, CN=root

        Validity

            Not Before: Nov 29 03:59:00 2018 GMT

            Not After : Nov 24 04:04:00 2038 GMT

        Subject: O=c0256a3b8e4b948bb9c21e66b0e1d9a72, OU=default,
CN=c0256a3b8e4b948bb9c21e66b0e1d9a72
```

访问

理解了原理之后，我们可以做一个简单的测试。我们以证书作为参数，使用 curl 访问 api server，并得到预期结果。

```
# curl --cert ./client.crt --cacert ./ca.crt --key ./client.key https://
xx.xx.xx.xxx:6443/api/

{

  "kind": "APIVersions",

  "versions": [

    "v1"

  ],

  "serverAddressByClientCIDRs": [

    {

      "clientCIDR": "0.0.0.0/0",

      "serverAddress": "192.168.0.222:6443"

    }

  ]

}
```

择优而居

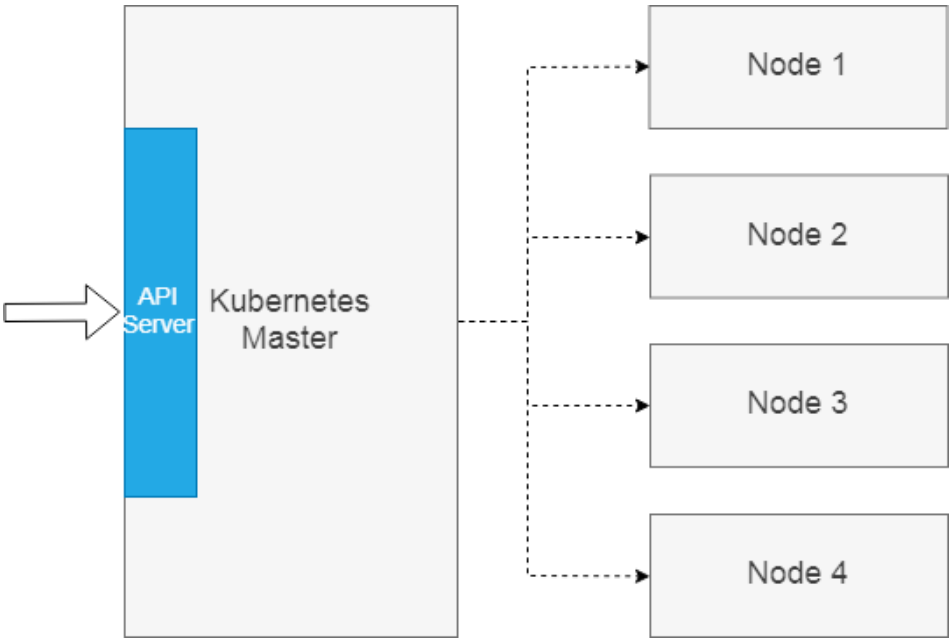
两种节点，一种任务

如开始所讲，Kubernetes 是管理集群多个节点的操作系统。这些节点在集群中的角色，却不必完全一样。Kubernetes 集群有两种节点，master 节点和 worker 节点。

这种角色的区分，实际上就是一种分工：master 负责整个集群的管理，其上运

行的以集群管理组件为主，这些组件包括实现集群入口的 api server；而 worker 节点主要负责承载普通任务。

在 Kubernetes 集群中，任务被定义为 pod 这个概念。pod 是集群可承载任务的原子单元。pod 被翻译成容器组，其实是意译，因为一个 pod 实际上封装了多个容器化的应用。原则上来讲，被封装在一个 pod 里边的容器，应该是存在相当程度的耦合关系。



择优而居

调度算法需要解决的问题，是替 pod 选择一个舒适的“居所”，让 pod 所定义的任务可以在这个节点上顺利地完成。

为了实现“择优而居”的目标，Kubernetes 集群调度算法采用了两步走的策略：第一步，从所有节点中排除不满足条件的节点，即预选；第二步，给剩余的节点打分，最后得分高者胜出，即优选。

下边，我们使用文章开始的时候制作的镜像，创建一个 pod，并通过日志来具体分析一下，这个 pod 怎么样被调度到某一个集群节点。

Pod 配置

首先，我们创建 pod 的配置文件，配置文件格式是 json。这个配置文件有三个地方比较关键，分别是镜像地址，命令以及容器的端口。

```
{
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
        "name": "app"
    },
    "spec": {
        "containers": [
            {
                "name": "app",
                "image": "registry.cn-hangzhou.aliyuncs.com/kube-easy/
app:latest",
                "command": [
                    "app"
                ],
                "ports": [
                    {
                        "containerPort": 2580
                    }
```

```
                ]
            }
        ]
    }
}
```

日志级别

集群调度算法被实现为运行在 master 节点上的系统组件，这一点和 api server 类似。其对应的进程名是 kube-scheduler。kube-scheduler 支持多个级别的日志输出，但社区并没有提供详细的日志级别说明文档。查看调度算法对节点进行筛选、打分的过程，我们需要把日志级别提高到 10，即加入参数 --v=10。

```
kube-scheduler --address=127.0.0.1 --kubeconfig=/etc/kubernetes/scheduler.
conf --leader-elect=true
--v=10
```

创建 Pod

使用 curl，以证书和 pod 配置文件等作为参数，通过 POST 请求访问 api server 的接口，我们可以在集群里创建对应的 pod。

```
# curl -X POST -H 'Content-Type: application/json;charset=utf-8' --cert ./
client.crt --cacert ./ca.crt --key
./client.key https://47.110.197.238:6443/api/v1/namespaces/default/pods -d@
app.json
```

预选

预选是 Kubernetes 调度的第一步，这一步要做的事情，是根据预先定义的规则，把不符合条件的节点过滤掉。不同版本的 Kubernetes 所实现的预选规则有很大的不同，但基本的趋势，是预选规则会越来越丰富。

比较常见的两个预选规则是 PodFitsResourcesPred 和 PodFitsHostPortsPred。前一个规则用来判断，一个节点上的剩余资源，是不是能够满足 pod 的需求；而后一个规则，检查一个节点上某一个端口是不是已经被其他 pod 所使用了。

下图是调度算法在处理测试 pod 的时候，输出的预选规则的日志。这段日志记录了预选规则 CheckVolumeBindingPred 的执行情况。某些类型的存储卷（PV），只能挂载到一个节点上，这个规则可以过滤掉不满足 pod 对 PV 需求的节点。

从 app 的编排文件里可以看到，pod 对存储卷并没有什么需求，所以这个条件并没有过滤掉节点。

```
16:31:46.097307 scheduler_binder.go        FindPodVolumes for pod "default/app"                    预选
                                           node "cn-hangzhou.i-bp1e5ayvk7vmtiutcttc"
16:31:46.097330 predicates.go              All PVCs found matches for pod default/app
                                           node "cn-hangzhou.i-bp1e5ayvk7vmtiutcttc"
16:31:46.097363 predicates.go              Schedule Pod default/app on Node is allowed
16:31:46.097394 scheduler_binder.go        FindPodVolumes for pod "default/app"
                                           node "cn-hangzhou.i-bp1e5ayvk7vmtiutcttd"
16:31:46.097429 predicates.go              All PVCs found matches for pod default/app
                                           node "cn-hangzhou.i-bp1e5ayvk7vmtiutcttd"
16:31:46.097443 predicates.go              Schedule Pod default/app on Node is allowed
16:31:46.097457 scheduler_binder.go        FindPodVolumes for pod "default/app"
                                           node "cn-hangzhou.i-bp1e5ayvk7vmtiutctte"
16:31:46.097466 predicates.go              All PVCs found matches for pod default/app
                                           node "cn-hangzhou.i-bp1e5ayvk7vmtiutctte"
16:31:46.097478 predicates.go              Schedule Pod default/app on Node is allowed
```

优选

调度算法的第二个阶段是优选阶段。这个阶段，kube-scheduler 会根据节点可用资源及其他一些规则，给剩余节点打分。

目前，CPU 和内存是调度算法考量的两种主要资源，但考量的方式并不是简单的，剩余 CPU、内存资源越多，得分就越高。

日志记录了两种计算方式：LeastResourceAllocation 和 BalancedResourceAllocation。前一种方式计算 pod 调度到节点之后，节点剩余 CPU 和内存占总 CPU 和内存的比例，比例越高得分就越高；第二种方式计算节点上 CPU 和内存使用比例之差的绝对值，绝对值越大，得分越少。

这两种方式，一种倾向于选出资源使用率较低的节点，第二种希望选出两种资源使用比例接近的节点。这两种方式有一些矛盾，最终依靠一定的权重来平衡这两个因素。

```
16:31:46.097681 resource_allocation.go     app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:                资源
                                           BalancedResourceAllocation
                                           capacity 4000 millicores 7153864704 memory bytes
                                           total request 6250 millicores 12983947264 memory bytes
                                           score 7
16:31:46.097693 resource_allocation.go     app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:
                                           LeastResourceAllocation
                                           capacity 4000 millicores 7153864704 memory bytes
                                           total request 6250 millicores 12983947264 memory bytes
                                           score 2
16:31:46.097702 resource_allocation.go     app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttd:
                                           BalancedResourceAllocation
                                           capacity 4000 millicores 7153864704 memory bytes
                                           total request 6375 millicores 13961220096 memory bytes
                                           score 6
16:31:46.097711 resource_allocation.go     app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttd:
                                           LeastResourceAllocation
                                           capacity 4000 millicores 7153864704 memory bytes
                                           total request 6375 millicores 13961220096 memory bytes
                                           score 2
16:31:46.097722 resource_allocation.go     app -> cn-hangzhou.i-bp1e5ayvk7vmtiutctte:
                                           BalancedResourceAllocation
                                           capacity 4000 millicores 7153864704 memory bytes
                                           total request 6355 millicores 13101387776 memory bytes
                                           score 7
16:31:46.097733 resource_allocation.go     app -> cn-hangzhou.i-bp1e5ayvk7vmtiutctte:
                                           LeastResourceAllocation
                                           capacity 4000 millicores 7153864704 memory bytes
                                           total request 6355 millicores 13101387776 memory bytes
                                           score 2
```

除了资源之外，优选算法会考虑其他一些因素，比如 pod 与节点的亲和性，或者如果一个服务有多个相同 pod 组成的情况下，多个 pod 在不同节点上的分散程度，这是保证高可用的一种策略。

```
16:31:46.097869 generic_scheduler.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:  其他
                                           TaintTolerationPriority, Score: (10)
16:31:46.097878 generic_scheduler.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttd:
                                           TaintTolerationPriority, Score: (10)
16:31:46.097882 generic_scheduler.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:
                                           TaintTolerationPriority, Score: (10)
16:31:46.097896 interpod_affinity.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:
                                           InterPodAffinityPriority, Score: (0)
16:31:46.097904 interpod_affinity.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttd:
                                           InterPodAffinityPriority, Score: (0)
16:31:46.097912 interpod_affinity.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:
                                           InterPodAffinityPriority, Score: (0)
16:31:46.097929 generic_scheduler.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:
                                           NodeAffinityPriority, Score: (0)
16:31:46.097937 generic_scheduler.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttd:
                                           NodeAffinityPriority, Score: (0)
16:31:46.097942 generic_scheduler.go       app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:
                                           NodeAffinityPriority, Score: (0)
16:31:46.097973 selector_spreading.go      app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:
                                           SelectorSpreadPriority, Score: (10)
16:31:46.097983 selector_spreading.go      app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttd:
                                           SelectorSpreadPriority, Score: (10)
16:31:46.097992 selector_spreading.go      app -> cn-hangzhou.i-bp1e5ayvk7vmtiutcttc:
                                           SelectorSpreadPriority, Score: (10)
```

得分

最后，调度算法会给所有的得分项乘以它们的权重，然后求和得到每个节点最终的得分。因为测试集群使用的是默认调度算法，而默认调度算法把日志中出现的得分项所对应的权重，都设置成了 1，所以如果按日志里有记录得分项来计算，最终三个节点的得分应该是 29,28 和 29。

```
16:31:46.098026 generic_scheduler.go       Host cn-hangzhou.i-bp1e5ayvk7vmtiutcttc => Score 100029   得分
16:31:46.098035 generic_scheduler.go       Host cn-hangzhou.i-bp1e5ayvk7vmtiutcttd => Score 100028
16:31:46.098040 generic_scheduler.go       Host cn-hangzhou.i-bp1e5ayvk7vmtiutcttc => Score 100029
```

之所以会出现日志输出的得分和我们自己计算的得分不符的情况，是因为日志并没有输出所有的得分项，猜测漏掉的策略应该是 NodePreferAvoidPodsPriority，这个策略的权重是 10000，每个节点得分 10，所以才得出最终日志输出的结果。

结束语

在这篇文章中，我们以一个简单的容器化 web 程序为例，着重分析了客户端怎么样通过 Kubernetes 集群 API Server 认证，以及容器应用怎么样被分派到合适节

点这两件事情。

在分析过程中，我们弃用了一些便利的工具，比如 kubectl，或者控制台。我们用了一些更接近底层的小实验，比如拆解 KubeConfig 文件，再比如分析调度器日志来分析认证和调度算法的运作原理。希望这些对大家进一步理解 Kubernetes 集群有所帮助。

集群服务的三个要点和一种实现

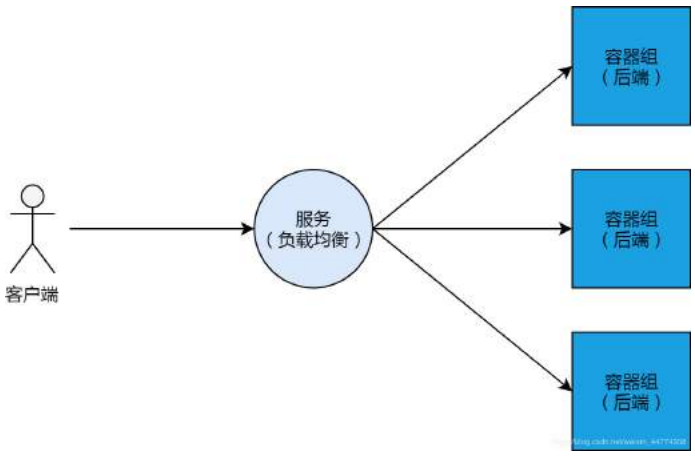
以我的经验来讲，理解 K8S 集群服务的概念，是比较不容易的一件事情。尤其是当我们基于似是而非的理解，去排查服务相关问题的时候，会非常不顺利。

这体现在，对于新手来说，ping 不通服务的 IP 地址这样基础的问题，都很难理解；而就算对经验很丰富的工程师来说，看懂服务相关的 iptables 配置，也是相当的挑战。

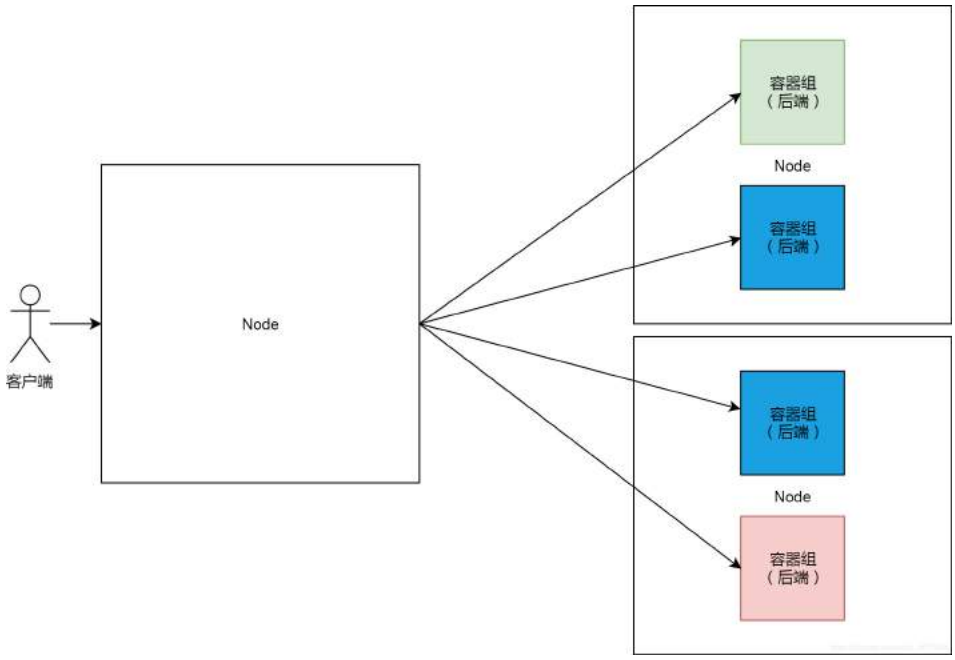
今天这边文章，我来深入解释一下 K8S 集群服务的原理与实现，便于大家理解。

K8S 集群服务的本质是什么

概念上来讲，K8S 集群的服务，其实就是负载均衡、或反向代理。这跟阿里云的负载均衡产品，有很多类似的地方。和负载均衡一样，服务有它的 IP 地址以及前端端口；服务后边会挂载多个容器组 Pod 作为其“后端服务器”，这些“后端服务器”有自己的 IP 以及监听端口。



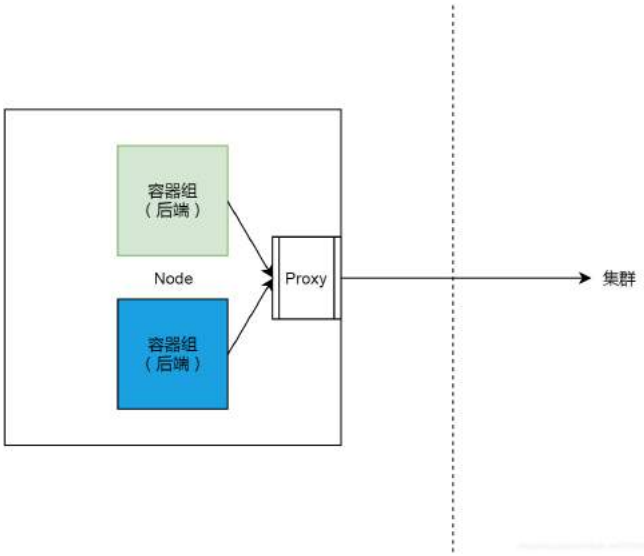
当这样的负载均衡和后端的架构，与 K8S 集群结合的时候，我们可以想到的最直观的实现方式，就是集群中某一个节点专门做负载均衡（类似 LVS）的角色，而其他节点则用来负载后端容器组。



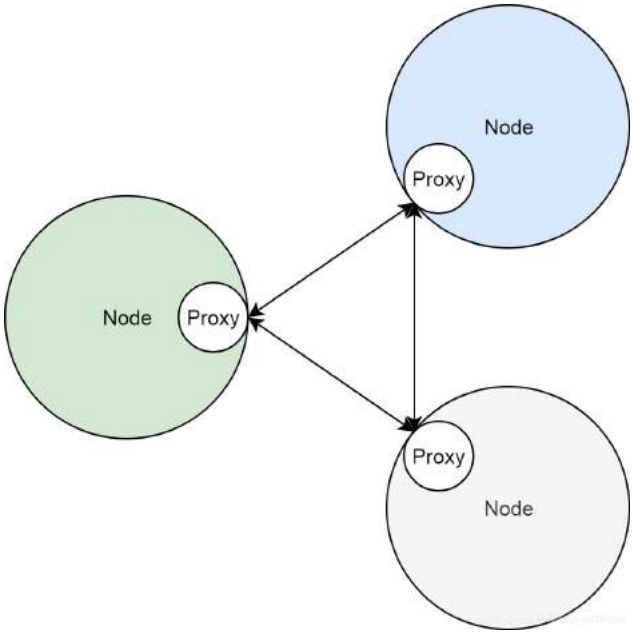
这样的实现方法，有一个巨大的缺陷，就是单点问题。K8S 集群是 Google 多年来自动化运维实践的结晶，这样的实现显然与其智能运维的哲学相背离的。

自带通信员

边车模式（Sidecar）是微服务领域的核心概念。边车模式，换一句通俗一点的说法，就是自带通信员。熟悉服务网格的同学肯定对这个很熟悉了。但是可能比较少人注意到，其实 K8S 集群原始服务的实现，也是基于 Sidecar 模式的。



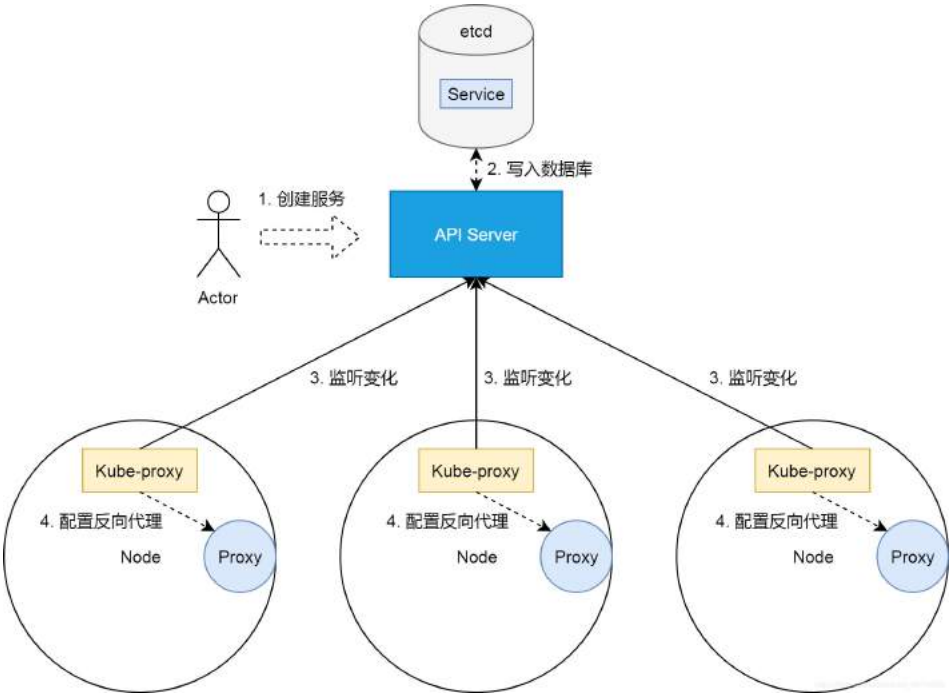
在 K8S 集群中，服务的实现，实际上是为每一个集群节点上，部署了一个反向代理 Sidecar。而所有对集群服务的访问，都会被节点上的反向代理转换成对服务后端容器组的访问。基本上来说，节点和这些 Sidecar 的关系如下图所示。



把服务照进现实

前边两节，我们看到了，K8S 集群的服务，本质上是负载均衡，即反向代理；同时我们知道了，在实际实现中，这个反向代理，并不是部署在集群某一个节点上，而是作为集群节点的边车，部署在每个节点上的。

在这里把服务照进反向代理这个现实的，是 K8S 集群的一个控制器，即 kube-proxy。关于 K8S 集群控制器的原理，请参考我另外一篇关于控制器的文章。简单来说，kube-proxy 作为部署在集群节点上的控制器，它们通过集群 API Server 监听着集群状态变化。当有新的服务被创建的时候，kube-proxy 则会把集群服务的状态、属性，翻译成反向代理的配置。



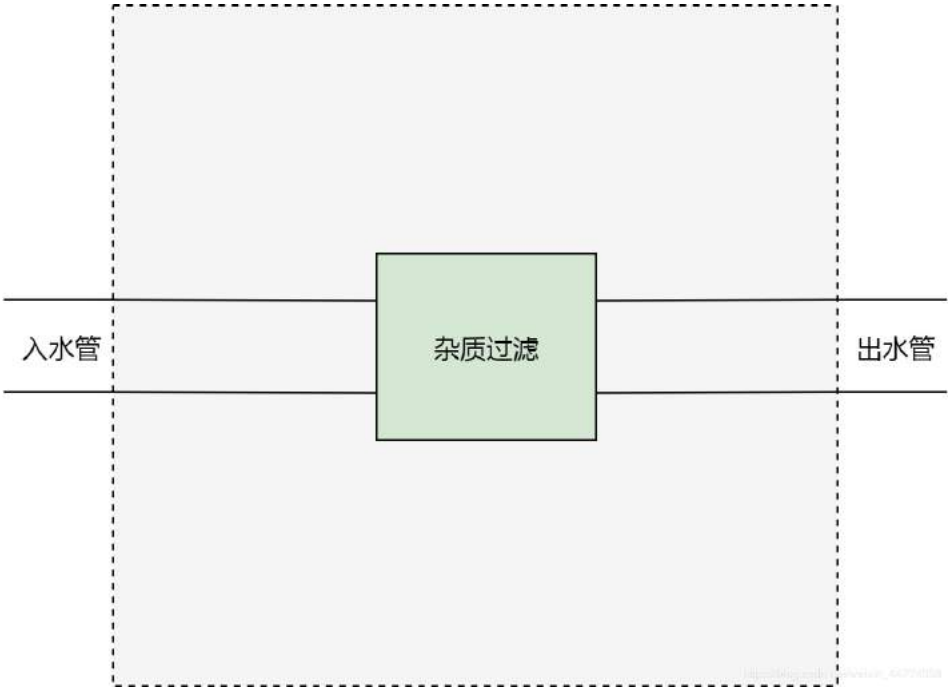
那剩下的问题，就是反向代理，即上图中 Proxy 的实现。

一种实现

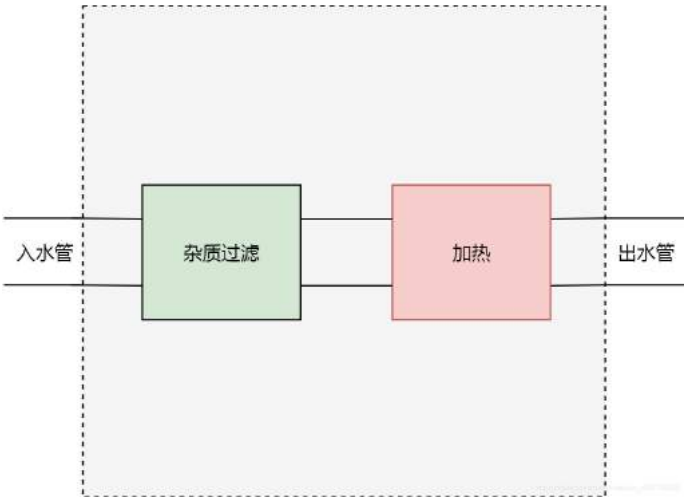
K8S 集群节点实现服务反向代理的方法，目前主要有三种，即 userspace、iptables 以及 ipvs。今天我们只深入分析 iptables 的方式，底层网络基于阿里云 flannel 集群网络。

过滤器框架

现在，我们来设想一种场景。我们有一个屋子。这个屋子有一个入水管和出水管。从入水管进入的水，是不能直接饮用的，因为有杂质。而我们期望，从出水管流出的水，可以直接饮用。为了达到目的，我们切开水管，在中间加一个杂质过滤器。

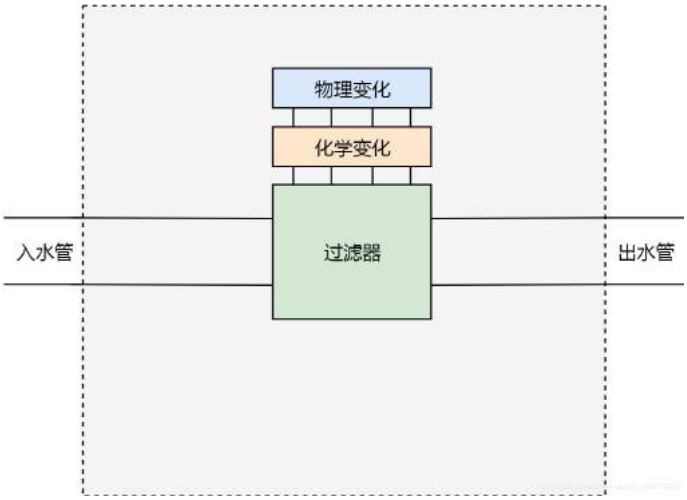


过了几天，我们的需求变了，我们不止要求从屋子里流出来的水可以直接饮用，我们还希望水是热水。所以我们不得不再在水管上增加一个切口，然后增加一个加热器。



很明显，这种切开水管，增加新功能的方式是很丑陋的。因为需求可能随时会变，我们甚至很难保证，在经过一年半载之后，这跟水管还能找得到可以被切开的地方。

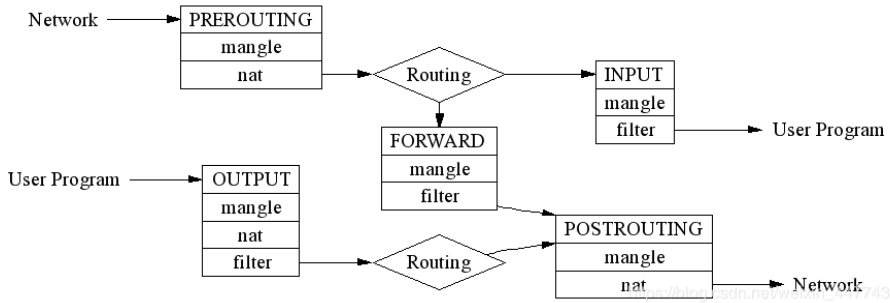
所以我们需要重新设计。首先我们不能随便切开水管，所以我们要把水管的切口固定下来。以上边的场景为例，我们确保水管只能有一个切口位置。其次，我们抽象出对水的两种处理方式：物理变化和化学变化。



基于以上的设计，如果我们需要过滤杂质，就可以在化学变化这个功能模块里增加一条过滤杂质的规则；如果我们需要增加温度的话，就可以在物理变化这个功能模块里增加一条加热的规则。

以上的过滤器框架，显然比切水管的方式，要优秀很多。设计这个框架，我们主要做了两件事情，一个是固定水管切口位置，另外一个是抽象出两种水处理方式。

理解这两件事情之后，我们可以来看下 iptables，或者更准确的说法，netfilter 的工作原理。netfilter 实际上就是一个过滤器框架。netfilter 在网络包收发及路由的管道上，一共切了 5 个口，分别是 PREROUTING，FORWARD，POSTROUTING，INPUT 以及 OUTPUT；同时 netfilter 定义了包括 nat、filter 在内的若干个网络包处理方式。



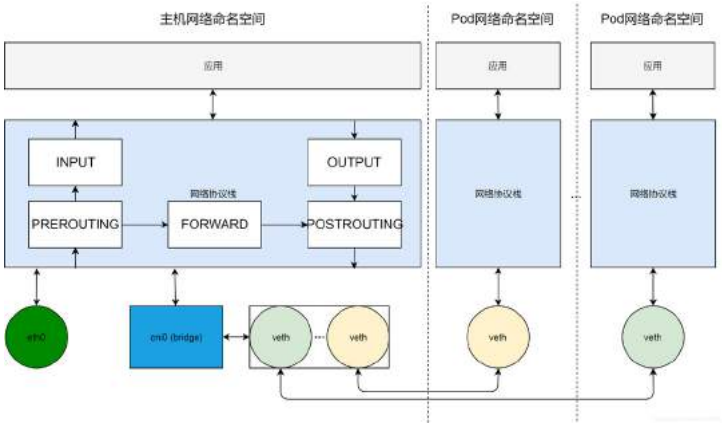
需要注意的是，routing 和 forwarding 很大程度上增加了以上 netfilter 的复杂程度，如果我们不考虑 routing 和 forwarding，那么 netfilter 会变得更我们的水质过滤器框架一样简单。

节点网络大图

现在我们看一下 K8S 集群节点的网络全貌。横向来看，节点上的网络环境，被分割成不同的网络命名空间，包括主机网络命名空间和 Pod 网络命名空间；纵向来看，每个网络命名空间包括完整的网络栈，从应用到协议栈，再到网络设备。

在网络设备这一层，我们通过 cni0 虚拟网桥，组建出系统内部的一个虚拟局域网。Pod 网络通过 veth 对连接到这个虚拟局域网内。cni0 虚拟局域网通过主机路由以及网口 eth0 与外部通信。

在网络协议栈这一层，我们可以通过编程 netfilter 过滤器框架，来实现集群节点的反向代理。



实现反向代理，归根结底，就是做 DNAT，即把发送给集群服务 IP 和端口的数据包，修改成发给具体容器组的 IP 和端口。

参考 netfilter 过滤器框架的图，我们知道，在 netfilter 里，可以通过在 PRE-ROUTING，OUTPUT 以及 POSTROUGING 三个位置加入 NAT 规则，来改变数据包的源地址或目的地址。

因为这里需要做的是 DNAT，即改变目的地址，这样的修改，必须在路由（ROUTING）之前发生以保证数据包可以被路由正确处理，所以实现反向代理的规则，需要被加到 PREROUTING 和 OUTPUT 两个位置。

其中，PREOURTING 的规则，用来处理从 Pod 访问服务的流量。数据包从 Pod 网络 veth 发送到 cni0 之后，进入主机协议栈，首先会经过 netfilter PREROUTING 来做处理，所以发给服务的数据包，会在这个位置做 DNAT。经过

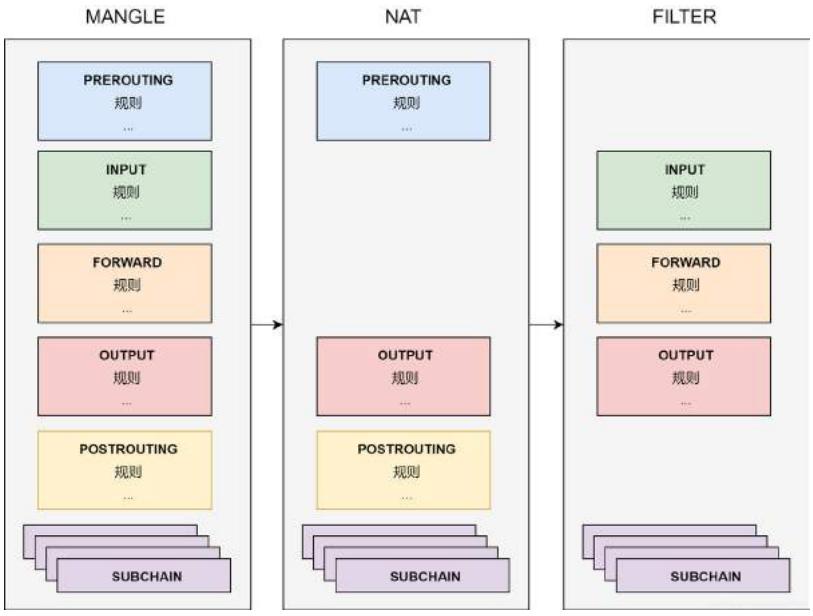
DNAT 处理之后，数据包的目的地址变成另外一个 Pod 的地址，从而经过主机路由，转发到 eth0，发送给正确的集群节点。

而添加在 OUTPUT 这个位置的 DNAT 规则，则用来处理从主机网络发给服务的数据包，原理也是类似，即经过路由之前，修改目的地址，以方便路由转发。

升级过滤器框架

在过滤器框架一节，我们看到 netfilter 是一个过滤器框架。netfilter 在数据“管到”上切了 5 个口，分别在这 5 个口上，做一些数据包处理工作。虽然固定切口位置以及网络包处理方式分类已经极大的优化了过滤器框架，但是有一个关键的问题，就是我们还是得在管道上做修改以满足新的功能。换句话说，这个框架没有做到管道和过滤功能两者的彻底解耦。

为了实现管道和过滤功能两者的解耦，netfilter 用了表这个概念。表就是 netfilter 的过滤中心，其核心功能是过滤方式的分类（表），以及每种过滤方式中，过滤规则的组织（链）。

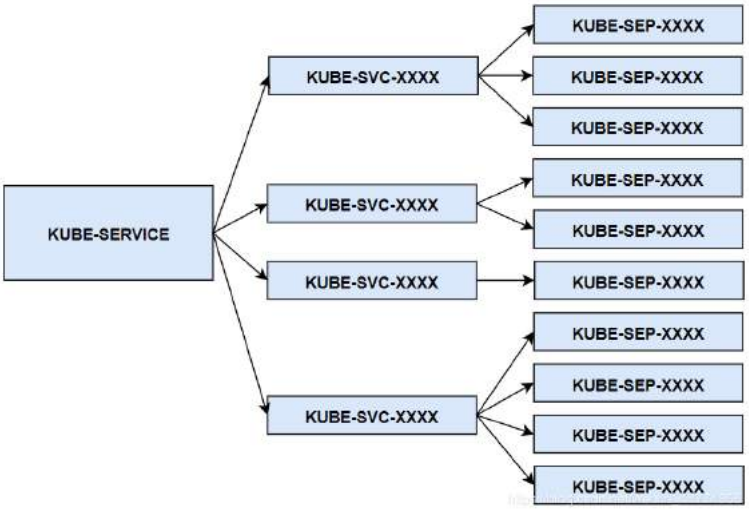


把过滤功能和管道解耦之后，所有对数据包的处理，都变成了对表的配置。而管道上的 5 个切口，仅仅变成了流量的出入口，负责把流量发送到过滤中心，并把处理之后的流量沿着管道继续传送下去。

如上图，在表中，netfilter 把规则组织成为链。表中有针对每个管道切口的默认链，也有我们自己加入的自定义链。默认链是数据的入口，默认链可以通过跳转到自定义链来完成一些复杂的功能。这里允许增加自定义链的好处是显然的。为了完成一个复杂过滤功能，比如实现 K8S 集群节点的反向代理，我们可以使用自定义链来模块化我们规则。

用自定义链实现服务的反向代理

集群服务的反向代理，实际上就是利用自定义链，模块化地实现了数据包的 DNAT 转换。KUBE-SERVICE 是整个反向代理的入口链，其对应所有服务的总入口；KUBE-SVC-XXXX 链是具体某一个服务的入口链，KUBE-SERVICE 链会根据服务 IP，跳转到具体服务的 KUBE-SVC-XXXX 链；而 KUBE-SEP-XXXX 链代表着某一个具体 Pod 的地址和端口，即 endpoint，具体服务链 KUBE-SVC-XXXX 会以一定算法（一般是随机），跳转到 endpoint 链。



而如前文中提到的，因为这里需要做的是 DNAT，即改变目的地址，这样的修改，必须在路由之前发生以保证数据包可以被路由正确处理。所以 KUBE-SERVICE 会被 PREROUTING 和 OUTPUT 两个默认链所调用。

总结

通过这篇文章，大家应该对 K8S 集群服务的概念以及实现，有了更深层次的认识。我们基本上需要把握三个要点。一、服务本质上是负载均衡；二、服务负载均衡的实现采用了与服务网格类似的 Sidecar 的模式，而不是 LVS 类型的独占模式；三、kube-proxy 本质上是一个集群控制器。除此之外，我们思考了过滤器框架的设计，并在此基础上，理解使用 iptables 实现的服务负载均衡的原理。

镜像拉取这件小事

导读：相比 K8s 集群的其他功能，私有镜像的自动拉取，看起来可能是比较简单的。而镜像拉取失败，大多数情况下都和权限有关。所以，在处理相关问题的时候，我们往往会轻松的说：这问题很简单，肯定是权限问题。但实际的情况是，我们经常为一个问题，花了多个人的时间却找不到原因。这主要还是我们对镜像拉取，特别是私有镜像自动拉取的原理理解不深。这篇文章，作者将带领大家讨论下相关原理。

顺序上来说，私有镜像自动拉取会首先通过阿里云 Acr credential helper 组件，再经过 K8s 集群的 API Server 和 kubelet 组件，最后到 docker 容器运行时。但是我的叙述，会从后往前，从最基本的 docker 镜像拉取说起。

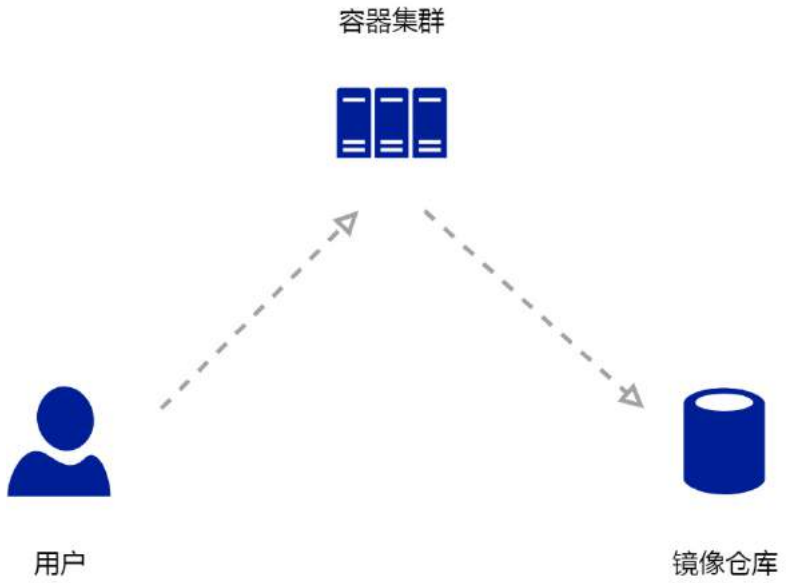
镜像拉取这件小事

为了讨论方便，我们来设想一个场景。很多人会使用网盘来存放一些文件，像照片，文档之类。当我们存取文件的时候，我们需要给网盘提供账户密码，这样网盘服务就能验证我们的身份。这时，我们是文件资源的所有者，而网盘则扮演着资源服务器的角色。账户密码作为认证方式，保证只有我们自己可以存取自己的文件。

这个场景足够简单，但很快我们就遇到新需求：我们需要使用一个在线制作相册

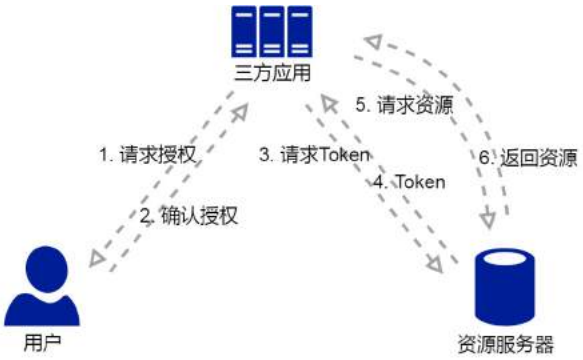
的应用。按正常的使用流程，我们需要把网盘的照片下载到本地，然后再把照片上传到电子相册。这个过程是比较很繁琐的。我们能想到的优化方法是，让相册应用，直接访问网盘来获取我们的照片，而这需要我们把用户名和密码授权给相册应用使用。

这样的授权方式，优点显而易见，但缺点也是很明显的：我们把网盘的用户名密码给了相册服务，相册服务就拥有了读写网盘的能力，从数据安全角度，这个是很可怕的。其实这是很多应用都会遇到的一个一般性场景。私有镜像拉取其实也是这个场景。这里的镜像仓库，就跟网盘一样，是资源服务器，而容器集群则是三方服务，它需要访问镜像仓库获取镜像。



理解 OAuth 2.0 协议

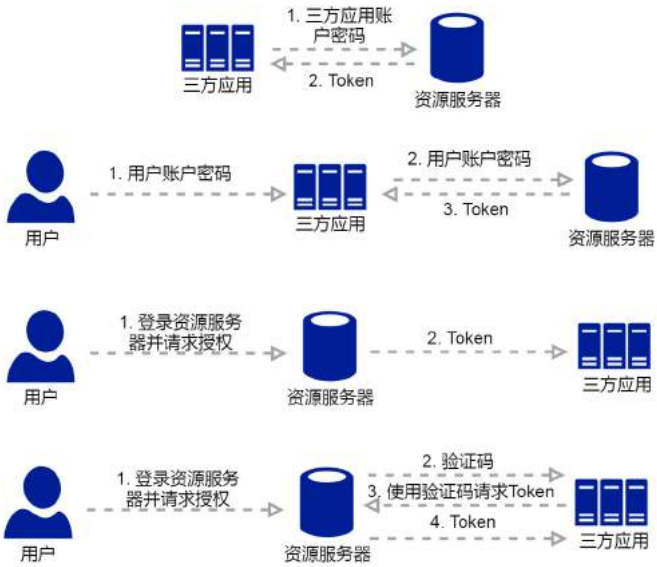
OAuth 协议是为了解决上述问题而设计的一种标准方案，我们的讨论针对 2.0 版本。相比把账户密码直接给三方应用，此协议采用了一种间接的方式来达到同样的目的。如下图，这个协议包括六个步骤，分别是三方应用获取用户授权，三方应用获取临时 Token 以及三方应用存取资源。



这六步理解起来不容易，主要是因为安全协议的设计，需要考虑协议的易证明性，所以我们换一种方式来解释这个协议。简单来说，这个协议其实就做了两件事情：

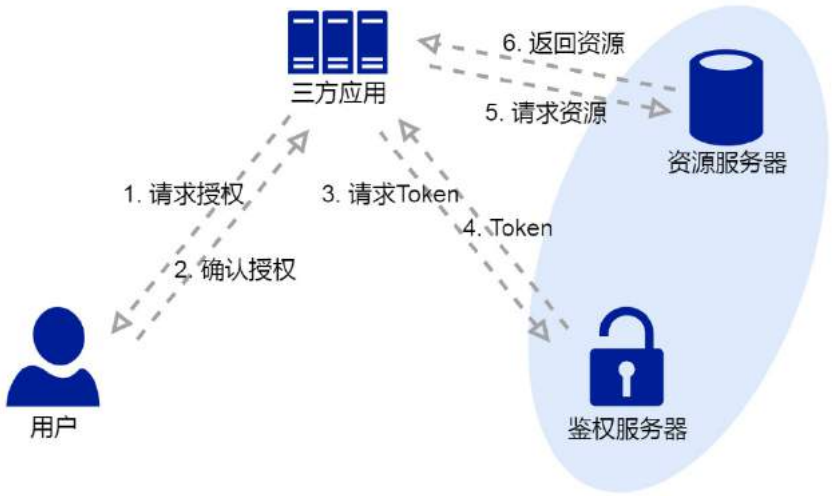
- 在用户授权的情况下，三方应用获取 token 所表示的临时访问权限；
- 然后三方应用使用这个 token 去获取资源。

如果用网盘的例子来说明的话，那就是用户授权网盘服务给相册应用创建临时 token，然后相册应用使用这个 token 去网盘服务获取用户的照片。实际上 OAuth 2.0 各个变种的核心差别，在于第一件事情，就是用户授权资源服务器的方式。



1. 最简单的一种，适用于三方应用本身就拥有被访问资源控制权限的情况。这种情况下，三方应用只需要用自己的账户密码登录资源服务器并申请临时 token 即可；
2. 当用户对三方应用足够信任的情况下，用户直接把账户密码给三方应用，三方应用使用账户密码向资源服务器申请临时 token；
3. 用户通过资源服务器提供的接口，登录资源服务器并授权资源服务器给三方应用发放 token；
4. 完整实现 OAuth 2.0 协议，也是最安全的。三方应用首先获取以验证码表示的用户授权，然后用此验证码从资源服务器换取临时 token，最后使用 token 存取资源。

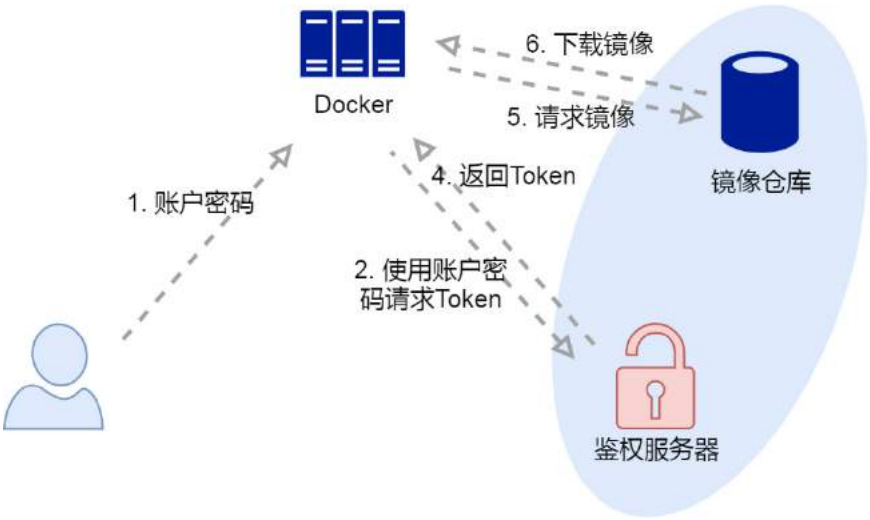
从上面的描述我们可以看到，资源服务器实际上扮演了鉴权和资源管理两种角色，这两者分开实现的话，协议流程会变成下图这样。



Docker 扮演的角色

大图

镜像仓库 Registry 的实现，目前使用“把账户密码给三方应用”的方式。即假设用户对 Docker 足够信任，用户直接将账户密码交给 Docker，然后 Docker 使用账户密码跟鉴权服务器申请临时 token。



理解 docker login

首先，我们在拉取私有镜像之前，要使用 docker login 命令来登录镜像仓库。这里的登录其实并没有和镜像仓库建立什么会话之类的关系。登录主要就做了三件事情：

- 第一件事情是跟用户要账户密码。

如下图，当执行登录命令，这个命会提示输入账户密码，这件事情对应的是大图的第一步。

```
[root@lab ~]# docker login registry.cn-shanghai.aliyuncs.com
Username: xxyyzz
Password:
```

- 第二件事情，docker 访问镜像仓库的 https 地址，并通过挑战 v2 接口来确认，接口是否会返回 Docker-Distribution-Api-Version 头字段。

这件事情在协议图中没有对应的步骤。它的作用跟 ping 差不多，只是确认下 v2 镜像仓库是否在线，以及版本是否匹配。

```
[root@lab ~]# curl https://registry.cn-shanghai.aliyuncs.com/v2/ -v
* About to connect() to registry.cn-shanghai.aliyuncs.com port 443 (#0)
*   Trying 139.196.71.17...
* Connected to registry.cn-shanghai.aliyuncs.com (139.196.71.17) port 443 (#0)
* Initializing NSS with certpath: sql:/etc/pki/nssdb
*   CAfile: /etc/pki/tls/certs/ca-bundle.crt
  CApath: none
* SSL connection using TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
* Server certificate:
*       subject: CN=*.registry.aliyuncs.com,O="Alibaba (China) Technology Co., Ltd.",L=HangZhou,ST=ZheJiang,C=CN
*       start date: Jan 28 03:01:05 2019 GMT
*       expire date: Jan 29 03:01:05 2020 GMT
*       common name: *.registry.aliyuncs.com
*       issuer: CN=GlobalSign Organization Validation CA - SHA256 - G2,O=GlobalSign nv-sa,C=BE
> GET /v2/ HTTP/1.1
> User-Agent: curl/7.29.0
> Host: registry.cn-shanghai.aliyuncs.com
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< Content-Type: application/json; charset=utf-8
< Docker-Distribution-Api-Version: registry/2.0
< Www-Authenticate: Bearer realm="https://dockerauth.cn-hangzhou.aliyuncs.com/auth",service="registry.aliyuncs.com:cn-shanghai:26842"
< Date: Mon, 23 Sep 2019 13:45:51 GMT
< Content-Length: 87
<
{"errors":[{"code":"UNAUTHORIZED","message":"authentication required","detail":null}]}
* Connection #0 to host registry.cn-shanghai.aliyuncs.com left intact
```

- 第三件事情，docker 使用用户提供的账户密码，访问 Www-Authenticate 头字段返回的鉴权服务器的地址 Bearer realm。

如果这个访问成功，则鉴权服务器会返回 jwt 格式的 token 给 docker，然后 docker 会把账户密码编码并保存在用户目录的 .docker/docker.json 文件里。

```
[root@lab ~]# curl https://dockerauth.cn-hangzhou.aliyuncs.com/auth -d grant_type=password -d username=                    -d password=
{"error":"incorrect username or password"}[root@lab ~]#
[root@lab ~]# curl https://dockerauth.cn-hangzhou.aliyuncs.com/auth -d grant_type=password -d username=                    -d password=
{"access_token":"

                "token":"

                                [root@lab ~]#
[root@lab ~]#
```

下图是我登录仓库之后的 docker.json 文件。这个文件作为 docker 登录仓库的唯一证据，在后续镜像仓库操作中，会被不断的读取并使用。其中关键信息 auth 就

是账户密码的 base64 编码。

```
[root@lab .docker]# cat config.json
{
        "auths": {
                "registry.cn-shanghai.aliyuncs.com": {
                        "auth": [redacted]
                }
        },
        "HttpHeaders": {
                "User-Agent": "Docker-Client/18.09.2 (linux)"
        }
}[root@lab .docker]#
```

拉取镜像是怎么回事

镜像一般会包括两部分内容，一个是 manifests 文件，这个文件定义了镜像的元数据，另一个是镜像层，是实际的镜像分层文件。镜像拉取基本上是围绕这两部分内容展开。因为我们这篇文章的重点是权限问题，所以我们这里只以 manifests 文件拉取为例。

拉取 manifests 文件，基本上也会做三件事情：

- 首先，docker 直接访问镜像 manifests 的地址，以便获取 Www-Authenticate 头字段。这个字段包括鉴权服务器的地址 Bearer realm，镜像服务地址 service，以及定义了镜像和操作的 scope。

```
[root@lab ~]# curl https://registry.cn-shanghai.aliyuncs.com/v2/debugging/busybox/manifests/latest -v
* About to connect() to registry.cn-shanghai.aliyuncs.com port 443 (#0)
*   Trying 139.196.71.17...
* Connected to registry.cn-shanghai.aliyuncs.com (139.196.71.17) port 443 (#0)
* Initializing NSS with certpath: sql:/etc/pki/nssdb
*   CAfile: /etc/pki/tls/certs/ca-bundle.crt
  CApath: none
* SSL connection using TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
* Server certificate:
*       subject: CN=*.registry.aliyuncs.com,O="Alibaba (China) Technology Co., Ltd.",L=HangZhou,ST=ZheJiang,C=CN
*       start date: Jan 28 03:01:05 2019 GMT
*       expire date: Jan 29 03:01:05 2020 GMT
*       common name: *.registry.aliyuncs.com
*       issuer: CN=GlobalSign Organization Validation CA - SHA256 - G2,O=GlobalSign nv-sa,C=BE
> GET /v2/debugging/busybox/manifests/latest HTTP/1.1
> User-Agent: curl/7.29.0
> Host: registry.cn-shanghai.aliyuncs.com
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< Content-Type: application/json; charset=utf-8
< Docker-Distribution-Api-Version: registry/2.0
< Www-Authenticate: Bearer realm="https://dockerauth.cn-hangzhou.aliyuncs.com/auth",service="registry.aliyuncs.com:cn-shanghai:26842",scope="repository:debugging/busybox:pull"
< Date: Mon, 23 Sep 2019 15:50:23 GMT
< Content-Length: 160
<
{"errors":[{"code":"UNAUTHORIZED","message":"authentication required","detail":[{"Type":"repository","Class":"","Name":"debugging/busybox","Action":"pull"}]}]}
```

- 接着，docker 访问上边拿到的 Bearer realm 地址来鉴权，以及在鉴权之后获取一个临时的 token。这对应协议大图使用账户密码获取临时 token 这一

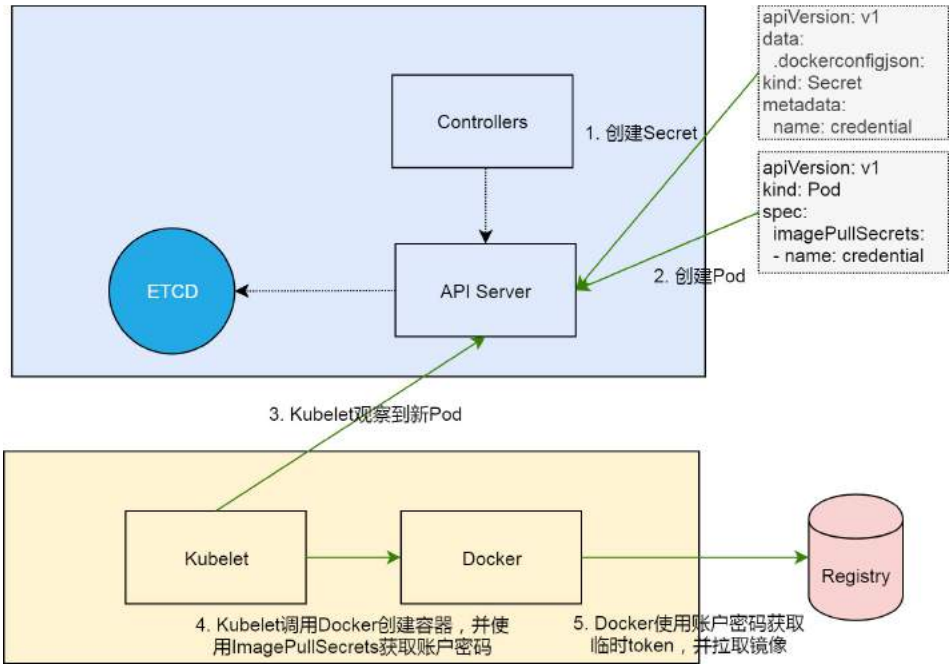
步，使用的账户密码直接读取自 docker.json 文件。

- 最后，使用上边的 token，以 Authorization 头字段的方式，来下载 manifests 文件。这对应的是协议大图下载镜像这一步。当然因为镜像还有分层文件，所以实际 docker 还会用这个临时 token 多次下载文件才能完整镜像下载。

K8s 实现的私有镜像自动拉取

基本功能

K8s 集群一般会管理多个节点，每个节点都有自己的 docker 环境。如果让用户分别到集群节点上登录镜像仓库，这显然是很不方便的。为了解决这个问题，K8s 实现了自动拉取镜像的功能。这个功能的核心，是把 docker.json 内容编码，并以 Secret 的方式作为 Pod 定义的一部分传给 Kubelet。



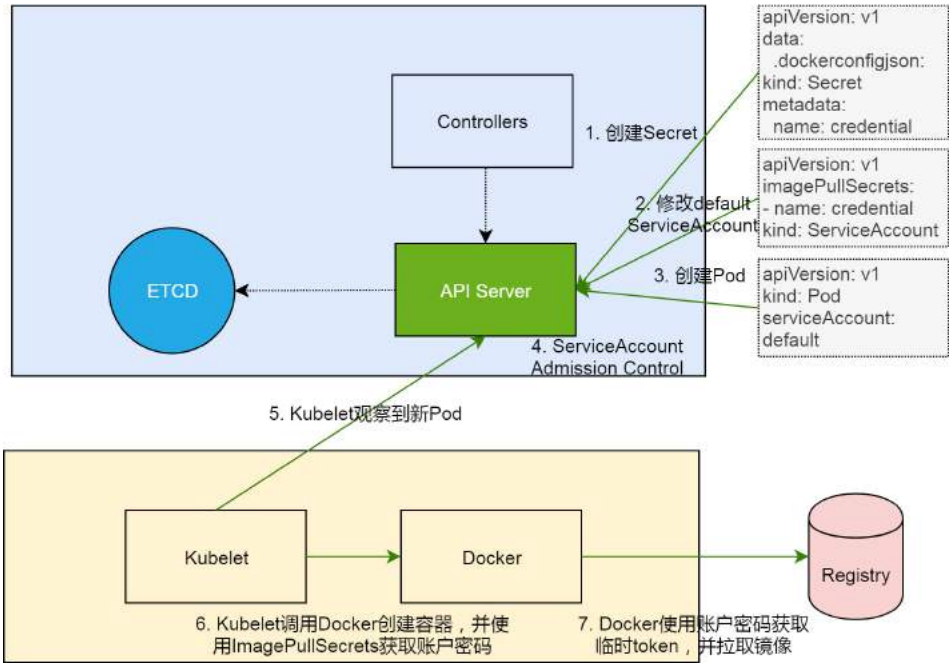
具体来说，步骤如下：

1. 创建 secret。这个 secret 的 .dockerconfigjson 数据项包括了一份 base64 编码的 docker.json 文件；
2. 创建 pod，且 pod 编排中 imagePullSecrets 指向第一步创建的 secret；
3. Kubelet 作为集群控制器，监控着集群的变化。当它发现新的 pod 被创建，就会通过 API Server 获取 pod 的定义，这包括 imagePullSecrets 引用的 secret；
4. Kubelet 调用 docker 创建容器且把 .dockerconfigjson 传给 docker；
5. 最后 docker 使用解码出来的账户密码拉取镜像，这和上一节的方法一致。

进阶方式

上边的功能，一定程度上解决了集群节点登录镜像仓库不方便的问题。但是我们在创建 Pod 的时候，仍然需要给 Pod 指定 imagePullSecrets。K8s 通过变更准入

控制（Mutating Admission Control）进一步优化了上边的基本功能。

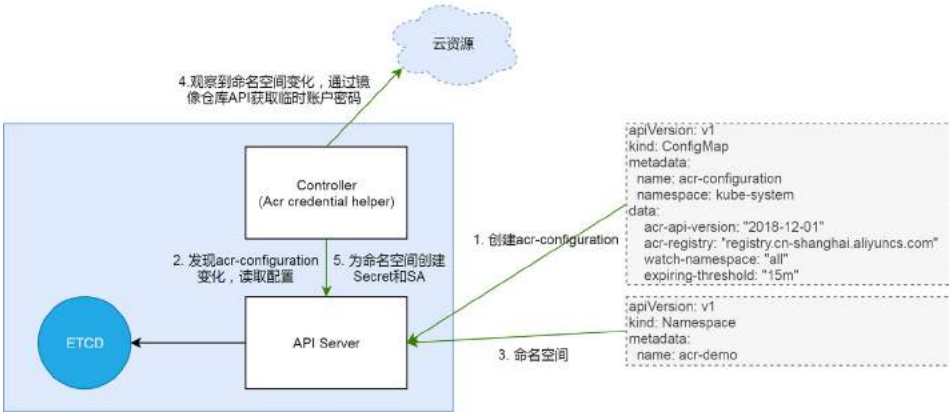


进一步优化的内容如下：

1. 在第一步创建 secret 之后，添加 default service account 对 imagePullSecrets 的引用；
2. Pod 默认使用 default service account，而 service account 变更准入控制器会在 default service account 引用 imagePullSecrets 的情况下，添加 imagePullSecrets 配置到 pod 的编排里。

阿里云实现的 Acr credential helper

阿里云容器服务团队，在 K8s 的基础上实现了控制器 Acr credential helper。这个控制器可以让同时使用阿里云 K8s 集群和容器镜像服务产品的用户，在不用配置自己账户密码的情况下，自动使用私有仓库中的容器镜像。



具体来说，控制器会监听 acr-configuration 这个 configmap 的变化，其主要关心 acr-registry 和 watch-namespace 这两个配置。前一个配置指定为临时账户授权的镜像仓库地址，后一个配置管理可以自动拉取镜像的命名空间。当控制器发现有命名空间需要被配置却没有被配置的时候，它会通过阿里云容器镜像服务的 API，来获取临时账户和密码。

有了临时账户密码，Acr credential helper 为命名空间创建对应的 Secret 以及更改 default SA 来引用这个 Secret。这样，控制器和 K8s 集群本身的功能，一起自动化了阿里云 K8s 集群拉取阿里云容器镜像服务上的镜像的全部流程。

总结

理解私有镜像自动拉取的实现，有一个难点和一个重点。

- **难点**是 OAuth 2.0 安全协议的原理，上文主要分析了为什么 OAuth 会这么设计；
- **重点**是集群控制器原理，因为整个自动化的过程，实际上是包括 Admission control 和 Acr credential helper 在内的多个控制器协作的结果。

实践篇

读懂这一篇，集群节点不下线

简介：排查完全陌生的问题，完全不熟悉的系统组件，是售后工程师的一大工作乐趣，当然也是挑战。今天借这篇文章，跟大家分析一例这样的问题。排查过程中，需要理解一些自己完全陌生的组件，比如 systemd 和 dbus。

排查完全陌生的问题，完全不熟悉的系统组件，是售后工程师的一大工作乐趣，当然也是挑战。今天借这篇文章，跟大家分析一例这样的问题。排查过程中，需要理解一些自己完全陌生的组件，比如 systemd 和 dbus。但是排查问题的思路和方法基本上还是可以复用了，希望对大家有所帮助。

问题一直在发生

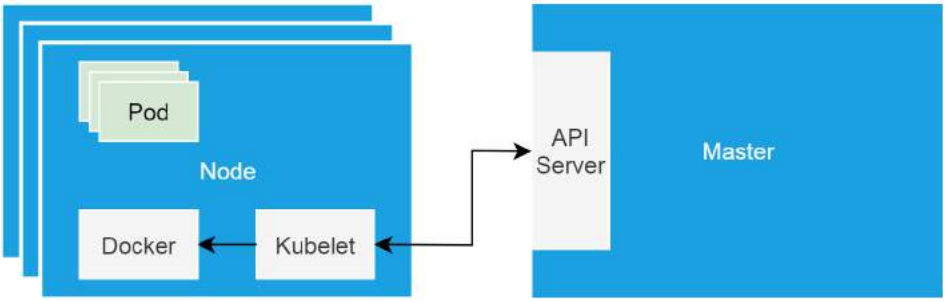
I'm NotReady

阿里云有自己的 Kubernetes 容器集群产品。随着 Kubernetes 集群出货量的剧增，线上用户零星的发现，集群会非常低概率地出现节点 NotReady 情况。据我们观察，这个问题差不多每个月，就会有一两个客户遇到。在节点 NotReady 之后，集群 Master 没有办法对这个节点做任何控制，比如下发新的 Pod，再比如抓取节点上正在运行 Pod 的实时信息。

```
[root@iZuf622u6us55fh7uo5nyeZ ~]# kubectl get nodes
NAME                                 STATUS     ROLES    AGE     VERSION
cn-shanghai.i-uf622u6us55fh7uo5nye   Ready      master   23d     v1.12.6-aliyun.1
cn-shanghai.i-uf68bayifvnr4kfds1uv   Ready      master   23d     v1.12.6-aliyun.1
cn-shanghai.i-uf68x08tpytogdtr7yj5   NotReady   <none>   23d     v1.12.6-aliyun.1
cn-shanghai.i-uf68x08tpytogdtr7yj6   Ready      <none>   23d     v1.12.6-aliyun.1
cn-shanghai.i-uf68x08tpytogdtr7yj7   Ready      <none>   4d14h   v1.12.6-aliyun.1
cn-shanghai.i-uf6alb4slwzoc78f70zv   Ready      master   23d     v1.12.6-aliyun.1
```

需要知道的 Kubernetes 知识

这里我稍微补充一点 Kubernetes 集群的基本知识。Kubernetes 集群的“硬件基础”，是以单机形态存在的集群节点。这些节点可以是物理机，也可以是虚拟机。集群节点分为 Master 和 Worker 节点。Master 节点主要用来负载集群管控组件，比如调度器和控制器。而 Worker 节点主要用来跑业务。Kubelet 是跑在各个节点上的代理，它负责与管控组件沟通，并按照管控组件的指示，直接管理 Worker 节点。



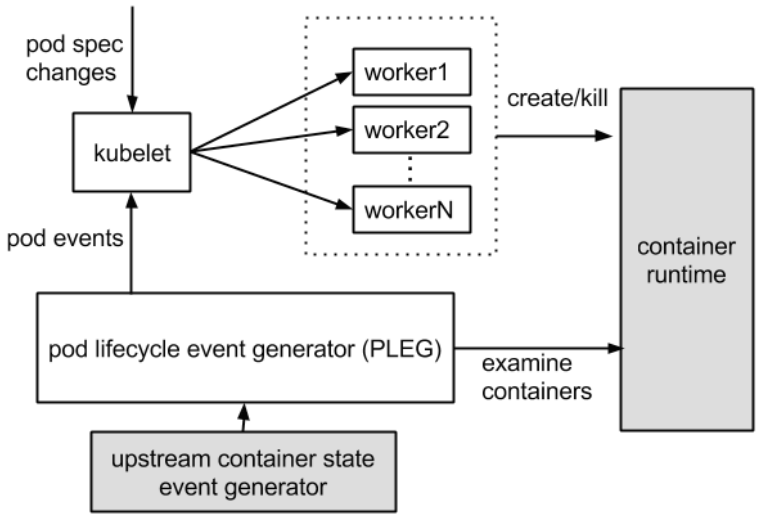
当集群节点进入 NotReady 状态的时候，我们需要做的第一件事情，肯定是检查运行在节点上的 kubelet 是否正常。在这个问题出现的时候，使用 systemctl 命令查看 kubelet 状态，发现它作为 systemd 管理的一个 daemon，是运行正常的。当我们用 journalctl 查看 kubelet 日志的时候，发现下边的错误。

```
[container runtime is down PLEG is not healthy: pleg was last seen active 6m5.000835156s ago; threshold is 3m0s]
[container runtime is down PLEG is not healthy: pleg was last seen active 6m10.001033976s ago; threshold is 3m0s]
[container runtime is down PLEG is not healthy: pleg was last seen active 6m15.001224916s ago; threshold is 3m0s]
[container runtime is down PLEG is not healthy: pleg was last seen active 6m20.001401078s ago; threshold is 3m0s]
[container runtime is down PLEG is not healthy: pleg was last seen active 6m25.001588667s ago; threshold is 3m0s]
[container runtime is down PLEG is not healthy: pleg was last seen active 6m30.001778732s ago; threshold is 3m0s]
[container runtime is down PLEG is not healthy: pleg was last seen active 6m35.001964418s ago; threshold is 3m0s]
[container runtime is down PLEG is not healthy: pleg was last seen active 6m40.002184419s ago; threshold is 3m0s]
[container runtime is down PLEG is not healthy: pleg was last seen active 6m45.002370225s ago; threshold is 3m0s]
```

什么是 PLEG

这个报错很清楚的告诉我们，容器 runtime 是不工作的，且 PLEG 是不健康的。这里容器 runtime 指的就是 docker daemon。Kubelet 通过直接操作 docker daemon 来控制容器的生命周期。而这里的 PLEG，指的是 pod lifecycle event

generator。PLEG 是 kubelet 用来检查容器 runtime 的健康检查机制。这件事情本来可以由 kubelet 使用 polling 的方式来做。但是 polling 有其成本上的缺陷，所以 PLEG 应用而生。PLEG 尝试以一种“中断”的形式，来实现对容器 runtime 的健康检查，虽然实际上，它同时用了 polling 和”中断”两种机制。

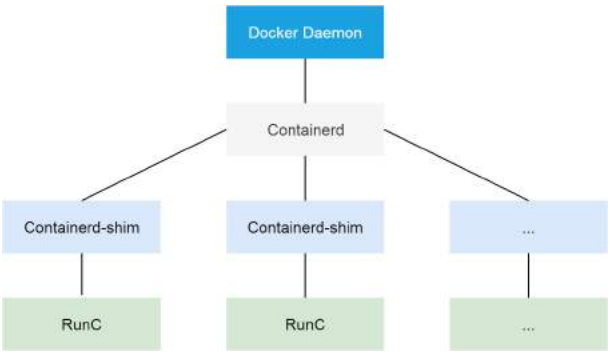


基本上看到上边的报错，我们可以确认，容器 runtime 出了问题。在有问题的节点上，通过 docker 命令尝试运行新的容器，命令会没有响应。这说明上边的报错是准确的 .

容器 runtime

Docker Daemon 调用栈分析

Docker 作为阿里云 Kubernetes 集群使用的容器 runtime，在 1.11 之后，被拆分成了多个组件以适应 OCI 标准。拆分之后，其包括 docker daemon，containerd，containerd-shim 以及 runC。组件 containerd 负责集群节点上容器的生命周期管理，并向上为 docker daemon 提供 gRPC 接口。

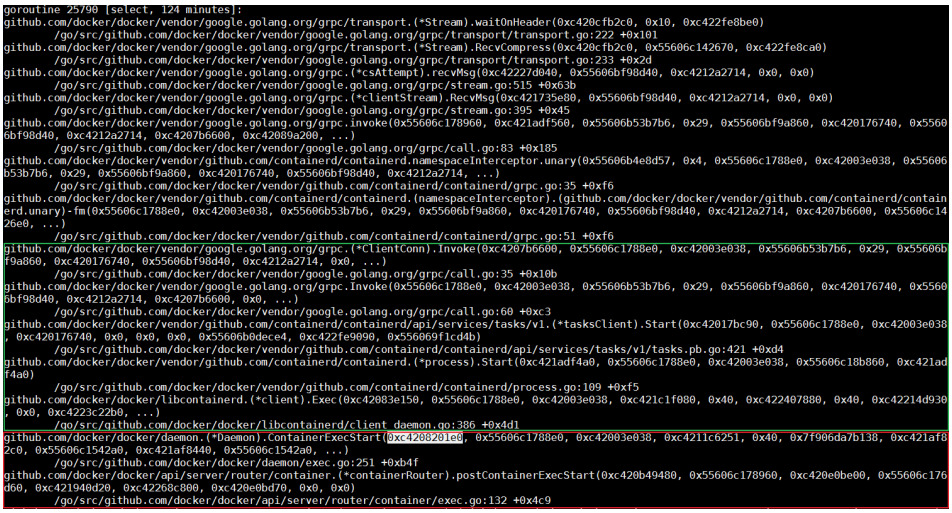


在这个问题中，既然 PLEG 认为容器运行是出了问题，我们需要先从 docker daemon 进程看起。我们可以使用 kill -USR1 <pid> 命令发送 USR1 信号给 docker daemon，而 docker daemon 收到信号之后，会把其所有线程调用栈输出到文件 /var/run/docker 文件夹里。

Docker daemon 进程的调用栈相对是比较容易分析的。稍微留意，我们会发现大多数的调用栈都类似下图中的样子。通过观察栈上每个函数的名字，以及函数所在的文件（模块）名称，我们可以看到，这个调用栈下半部分，是进程接到 http 请求，做请求路由的过程；而上半部分则进入实际的处理函数。最终处理函数进入等待状态，等待的是一个 mutex 实例。

```
goroutine 29184 [semacquire, 112 minutes]:
sync.runtime_SemacquireMutex(0xc421b30084, 0x0)
        /usr/local/go/src/runtime/sema.go:71 +0x3f
sync.(*Mutex).Lock(0xc421b30080)
        /usr/local/go/src/sync/mutex.go:134 +0x10a
github.com/docker/docker/daemon.(*Daemon).ContainerInspectCurrent(0xc4208201e0, 0xc420cc4be6, 0x48, 0x0, 0xc422431580, 0xc422431560, 0xc422bc5e90)
        /go/src/github.com/docker/docker/daemon/inspect.go:80 +0xbe
github.com/docker/docker/daemon.(*Daemon).ContainerInspect(0xc4208201e0, 0xc420cc4be6, 0x48, 0x0, 0xc420cc4bd6, 0x4, 0x55606bd4e5c0, 0xc4221cd801, 0xc
4208dfe40, 0xc4221cd7c8)
        /go/src/github.com/docker/docker/daemon/inspect.go:29 +0x1id
github.com/docker/docker/api/server/router/container.(*containerRouter).getContainersByName(0xc428b49480, 0x55606c178960, 0xc422fce150, 0x55606c176d60
, 0xc422ba8388, 0xc420d91300, 0xc422fce0c0, 0x55606b4e9cfb, 0x5)
        /go/src/github.com/docker/docker/api/server/router/container/inspect.go:15 +0x119
github.com/docker/docker/api/server/router/container.(*containerRouter).(github.com/docker/docker/api/server/router/container.getContainersByName)-fm(
0x55606c178960, 0xc422fce150, 0x55606c176d60, 0xc422ba8388, 0xc420d01300, 0xc422fce0c0, 0x55606a82a7cc, 0x55606bfb48e0)
        /go/src/github.com/docker/docker/api/server/router/container/container.go:39 +0x6b
github.com/docker/docker/api/server/middleware.ExperimentalMiddleware.WrapHandler.func1(0x55606c178960, 0xc422fce150, 0x55606c176d60, 0xc422ba8388, 0x
c420d91300, 0xc422fce0c0, 0x55606c178960, 0xc422fce150)
        /go/src/github.com/docker/docker/api/server/middleware/experimental.go:26 +0xda
github.com/docker/docker/api/server/middleware.VersionMiddleware.WrapHandler.func1(0x55606c178960, 0xc422fce120, 0x55606c176d60, 0xc422ba8388, 0xc420d
91300, 0xc422fce0c0, 0x0, 0xc4221cda58)
        /go/src/github.com/docker/docker/api/server/middleware/version.go:62 +0x401
github.com/docker/docker/pkg/authorization.(*Middleware).WrapHandler.func1(0x55606c178960, 0xc422fce120, 0x55606c176d60, 0xc422ba8388, 0xc420d91300, 0
xc422fce0c0, 0x55606c178960, 0xc422fce120)
        /go/src/github.com/docker/docker/pkg/authorization/middleware.go:59 +0x7ab
github.com/docker/docker/api/server.(*Server).makeHTTPHandler.func1(0x55606c176d60, 0xc422ba8380, 0xc420d91300)
        /go/src/github.com/docker/docker/api/server/server.go:141 +0x19a
net/http.HandlerFunc.ServeHTTP(0xc420c764e0, 0x55606c176d60, 0xc422ba8380, 0xc420d91300)
        /usr/local/go/src/net/http/server.go:1947 +0x46
github.com/docker/docker/vendor/github.com/gorilla/mux.(*Router).ServeHTTP(0xc420cbj040, 0x55606c176d60, 0xc422ba8380, 0xc420d91300)
        /go/src/github.com/docker/docker/vendor/github.com/gorilla/mux/mux.go:103 +0x228
github.com/docker/docker/api/server.(*routerSwapper).ServeHTTP(0xc420fc7cd0, 0x55606c176d60, 0xc422ba8380, 0xc420d91300)
        /go/src/github.com/docker/docker/api/server/router_swapper.go:29 +0x72
net/http.serverHandler.ServeHTTP(0xc420880040, 0x55606c176d60, 0xc422ba8380, 0xc420d91300)
        /usr/local/go/src/net/http/server.go:2697 +0xbe
net/http.(*conn).serve(0xc42187c5a0, 0x55606c1788a0, 0xc420ad8e40)
        /usr/local/go/src/net/http/server.go:1830 +0x653
created by net/http.(*Server).Serve
        /usr/local/go/src/net/http/server.go:2798 +0x27d
```

到这里，我们需要稍微看一下 ContainerInspectCurrent 这个函数的实现，而最重要的是，我们能搞明白，这个函数的第一个参数，就是 mutex 的指针。使用这个指针搜索整个调用栈文件，我们会找出，所有等在这个 mutex 上边的线程。同时，我们可以看到下边这个线程。

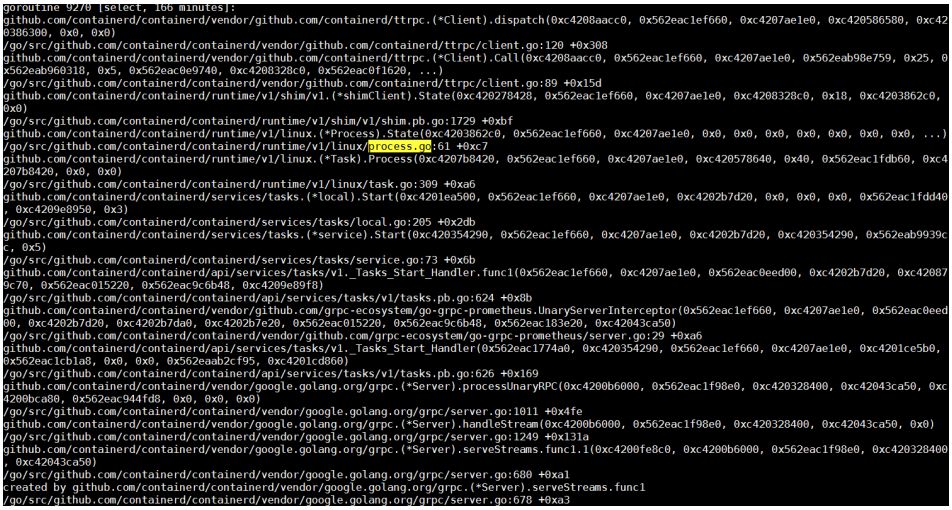


这个线程上，函数 ContainerExecStart 也是在处理具体请求的时候，收到了这个 mutex 这个参数。但不同的是，ContainerExecStart 并没有在等待 mutex，而是已经拿到了 mutex 的所有权，并把执行逻辑转向了 containerd 调用。关于这一点，我们可以使用代码来验证。前边我们提到过，containerd 向上通过 gRPC 对 docker daemon 提供接口。此调用栈上半部分内容，正是 docker daemon 在通过 gRPC 请求来呼叫 containerd。

Containerd 调用栈分析

与输出 docker daemon 的调用栈类似，我们可以通过 kill −SIGUSR1 <pid> 命令来输出 containerd 的调用栈。不同的是，这次调用栈会直接输出到 messages 日志。

Containerd 作为一个 gRPC 的服务器，它会在接到 docker daemon 的远程请求之后，新建一个线程去处理这次请求。关于 gRPC 的细节，我们这里其实不用关注太多。在这次请求的客户端调用栈上，可以看到这次调用的核心函数是 Start 一个进程。我们在 containerd 的调用栈里搜索 Start，Process 以及 process.go 等字段，很容易发现下边这个线程。



这个线程的核心任务，就是依靠 runC 去创建容器进程。而在容器启动之后，runC 进程会退出。所以下一步，我们自然而然会想到，runC 是不是有顺利完成自己的任务。查看进程列表，我们会发现，系统中有个别 runC 进程，还在执行，这不是预期内的行为。容器的启动，跟进程的启动，耗时应该是差不对的，系统里有正在运行的 runC 进程，则说明 runC 不能正常启动容器。

什么是 Dbus

RunC 请求 Dbus

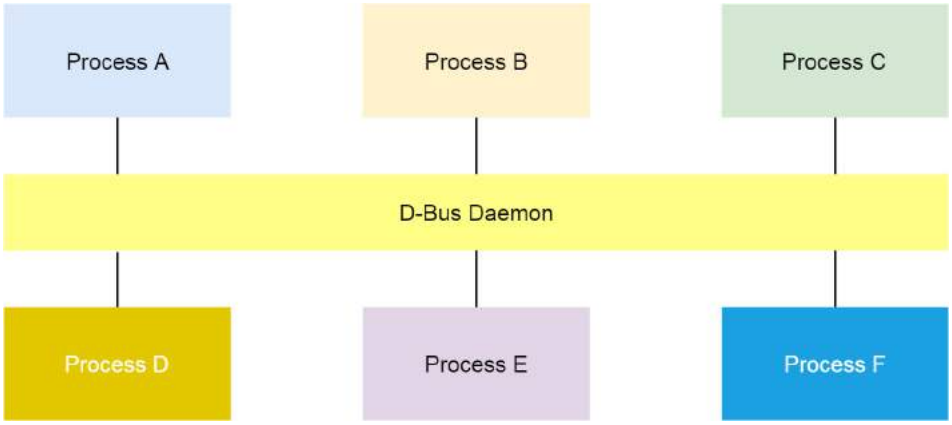
容器 runtime 的 runC 命令，是 libcontainer 的一个简单的封装。这个工具可以用来管理单个容器，比如容器创建，或者容器删除。在上节的最后，我们发现 runC

不能完成创建容器的任务。我们可以把对应的进程杀掉，然后在命令行用同样的命令尝试启动容器，同时用 strace 追踪整个过程。

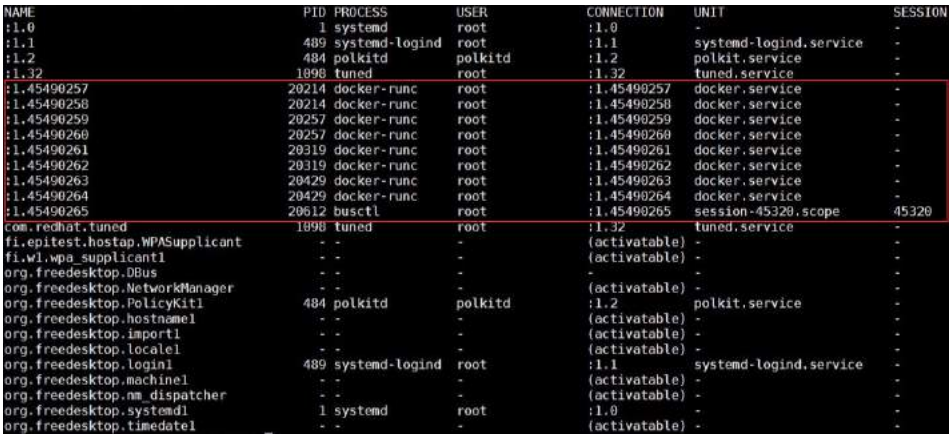
```
socket(AF_LOCAL, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, 0) = 5
setsockopt(5, SOL_SOCKET, SO_BROADCAST, [1], 4) = 0
connect(5, {sa_family=AF_LOCAL, sun_path="/var/run/dbus/system_bus_socket"}, 34) = 0
epoll_ctl(4, EPOLL_CTL_ADD, 5, {EPOLLIN|EPOLLOUT|EPOLLRDHUP|EPOLLET, {u32=94518944, u64=140630208691872}}) = 0
getsockname(5, {sa_family=AF_LOCAL, NULL}, [2]) = 0
getpeername(5, {sa_family=AF_LOCAL, sun_path="/run/dbus/system_bus_socket"}, [30]) = 0
getuid()                                = 0
getpid()                                = 25793
getuid()                                = 0
getgid()                                = 0
getsockopt(5, SOL_SOCKET, SO_TYPE, [1], [4]) = 0
sendmsg(5, {msg_name(0)=NULL, msg_iov(1)=[{"\0", 1}], msg_controllen=32, [{cmsg_len=28, cmsg_level=SOL_SOCKET, cmsg_type=SCM_CREDENTIALS, {pid=25793,
uid=0, gid=0}}], msg_flags=0}, 0) = 1
write(5, "AUTH\r\n", 6)                 = 6
read(5, "REJECTED EXTERNAL DBUS_COOKIE_SH"..., 4096) = 46
write(5, "AUTH EXTERNAL 30\r\n", 18)    = 18
read(5, "OK 720b293370de55e8587dda875cbb4"..., 4096) = 37
write(5, "NEGOTIATE_UNIX_FD\r\n", 19)   = 19
read(5, "AGREE_UNIX_FD\r\n", 4096)      = 15
write(5, "BEGIN\r\n", 7)                = 7
futex(0xc420034948, FUTEX_WAKE, 1)      = 1
write(5, "l\1\0\1\0\0\0\0\1\0\0\0n\0\0\0\2\1s\0\24\0\0\0org.free"..., 128) = 128
futex(0x5635a2de5a48, FUTEX_WAIT, 0, NULL) = 0
futex(0x5635a2de5a48, FUTEX_WAIT, 0, NULL) = 0
epoll_pwait(4, [], 128, 0, NULL, 0)     = 0
```

分析发现，runC 停在了向带有 org.free 字段的 dbus 写数据的地方。那什么是 dbus 呢？在 Linux 上，dbus 是一种进程间进行消息通信的机制。

原因并不在 Dbus



我们可以使用 busctl 命令列出系统现有的所有 bus。如下图，在问题发生的时候，我看到客户集群节点 Name 的编号非常大。所以我倾向于认为，dbus 某些相关的数据结构，比如 Name，耗尽了引起了这个问题。



```
NAME                                PID PROCESS         USER             CONNECTION    UNIT                         SESSION
:1.0                                  1 systemd         root             :1.0          -                            -
:1.1                                489 systemd-logind  root             :1.1          systemd-logind.service       -
:1.2                                484 polkitd         polkitd          :1.2          polkit.service               -
:1.32                              1098 tuned           root             :1.32         tuned.service                -
:1.45490257                       20214 docker-runc     root             :1.45490257   docker.service               -
:1.45490258                       20214 docker-runc     root             :1.45490258   docker.service               -
:1.45490259                       20257 docker-runc     root             :1.45490259   docker.service               -
:1.45490260                       20257 docker-runc     root             :1.45490260   docker.service               -
:1.45490261                       20319 docker-runc     root             :1.45490261   docker.service               -
:1.45490262                       20319 docker-runc     root             :1.45490262   docker.service               -
:1.45490263                       20429 docker-runc     root             :1.45490263   docker.service               -
:1.45490264                       20429 docker-runc     root             :1.45490264   docker.service               -
:1.45490265                       20612 busctl          root             :1.45490265   session-45320.scope          45320
com.redhat.tuned                   1098 tuned           root             :1.32         tuned.service                -
fi.epitest.hostap.WPASupplicant       - -               -                (activatable) -                            -
fi.w1.wpa_supplicant1                 - -               -                (activatable) -                            -
org.freedesktop.DBus                  - -               -                -             -                            -
org.freedesktop.NetworkManager        - -               -                (activatable) -                            -
org.freedesktop.PolicyKit1          484 polkitd         polkitd          :1.2          polkit.service               -
org.freedesktop.hostname1             - -               -                (activatable) -                            -
org.freedesktop.import1               - -               -                (activatable) -                            -
org.freedesktop.locale1               - -               -                (activatable) -                            -
org.freedesktop.login1              489 systemd-logind  root             :1.1          systemd-logind.service       -
org.freedesktop.machine1              - -               -                (activatable) -                            -
org.freedesktop.nm_dispatcher         - -               -                (activatable) -                            -
org.freedesktop.systemd1              1 systemd         root             :1.0          -                            -
org.freedesktop.timedate1             - -               -                (activatable) -                            -
```

Dbus 机制的实现，依赖于一个组件叫做 dbus-daemon。如果真的是 dbus 相关数据结构耗尽，那么重启这个 daemon，应该是可以解决这个问题。但不幸的是，问题并没有这么直接。重启 dbus-daemon 之后，问题依然存在。

在上边用 strace 追踪 runC 的截图中，我提到了，runC 卡在向带有 org.free 字段的 bus 写数据的地方。在 busctl 输出的 bus 列表里，显然带有这个字段的 bus，都在被 systemd 使用。这时，我们用 systemctl daemon-reexec 来重启 systemd，问题消失了。所以基本上我们可以判断一个方向：问题可能跟 systemd 有关系。

Systemd 是硬骨头

Systemd 是相当复杂的一个组件，尤其对没有做过相关开发工作的同学来说，比如我自己。基本上，排查 systemd 的问题，我用到了四个方法，（调试级别）日志，core dump，代码分析，以及 live debugging。其中第一个，第三个和第四个结合起来使用，让我在经过几天的鏖战之后，找到了问题的原因。但是这里我们先从“没用”的 core dump 说起。

没用的 Core Dump

因为重启 systemd 解决了问题，而这个问题本身，是 runC 在使用 dbus 和 systemd 通信的时候没有了响应，所以我们需要验证的第一件事情，就是 systemd 不是有关键线程被锁住了。查看 core dump 里所有线程，只有以下一个线程，此线程并没有被锁住，它在等待 dbus 事件，以便做出响应。

```
(gdb) thread apply all bt

Thread 1 (Thread 0x7f4b1d168940 (LWP 1)):
#0  0x00007f4b1b992463 in __epoll_wait_nocancel () from /lib64/libc.so.6
#1  0x000055a2317ffa69 in sd_event_wait (e=e@entry=0x55a2333e54f0, timeout=timeout@entry=18446744073709551615)
    at src/libsystemd/sd-event/sd-event.c:2372
#2  0x000055a23180057d in sd_event_run (e=0x55a2333e54f0, timeout=18446744073709551615) at src/libsystemd/sd-event/sd-event.c:2499
#3  0x000055a2317610c3 in manager_loop (m=0x55a2333e5050) at src/core/manager.c:2252
#4  0x000055a2317555fb in main (argc=5, argv=0x7ffdd8e7f828) at src/core/main.c:1773
```

零散的信息

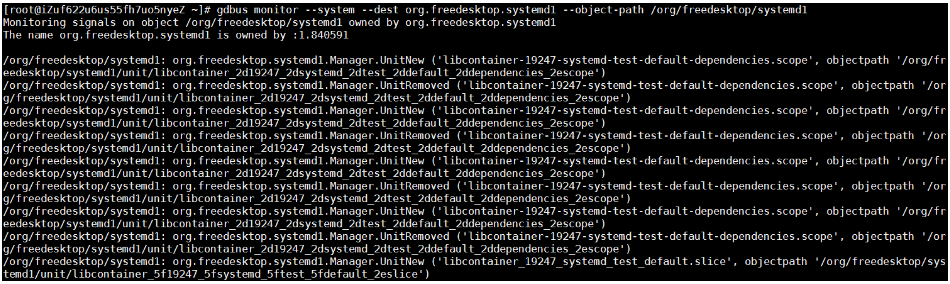
因为无计可施，所以只能做各种测试、尝试。使用 busctl tree 命令，可以输出所有 bus 上对外暴露的接口。从输出结果看来，org.freedesktop.systemd1 这个 bus 是不能响应接口查询请求的。

```
[root@iz2ze57isp1al1llyvapgkz log]# busctl tree
Service org.freedesktop.DBus:
Only root object discovered.

Service org.freedesktop.login1:
└─/org/freedesktop/login1
  ├─/org/freedesktop/login1/seat
  │ └─/org/freedesktop/login1/seat/seat0
  ├─/org/freedesktop/login1/session
  │ └─/org/freedesktop/login1/session/_39273
  └─/org/freedesktop/login1/user
    └─/org/freedesktop/login1/user/_0

Service org.freedesktop.systemd1:
Failed to introspect object / of service org.freedesktop.systemd1: Connection timed out
No objects discovered.

Service org.freedesktop.PolicyKit1:
└─/org
  └─/org/freedesktop
    └─/org/freedesktop/PolicyKit1
      └─/org/freedesktop/PolicyKit1/Authority

Service com.redhat.tuned:
└─/Tuned
```

使用下边的命令，观察 org.freedesktop.systemd1 上接受到的所以请求，可以看到，在正常系统里，有大量 Unit 创建删除的消息，但是有问题的系统里，这个

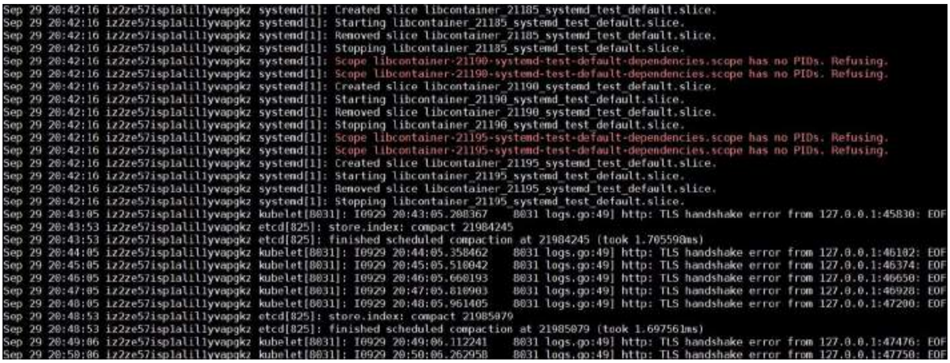
bus 上完全没有任何消息。

gdbus monitor --system --dest org.freedesktop.systemd1 --object-path /org/freedesktop/systemd1



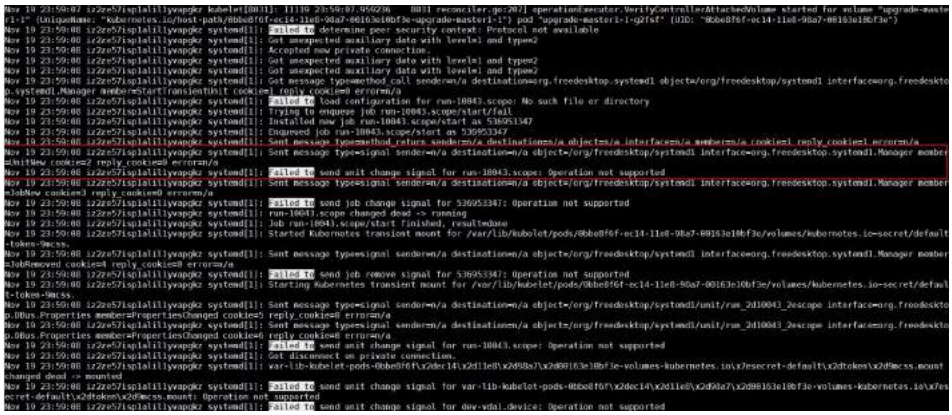
```
[root@iZuf622u6us55fh7uo5nyeZ ~]# gdbus monitor --system --dest org.freedesktop.systemd1 --object-path /org/freedesktop/systemd1
Monitoring signals on object /org/freedesktop/systemd1 owned by org.freedesktop.systemd1
The name org.freedesktop.systemd1 is owned by :1.840591

/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitNew ('libcontainer-19247-systemd-test-default-dependencies.scope', objectpath '/org/freedesktop/systemd1/unit/libcontainer_2d19247_2dsystemd_2dtest_2ddefault_2ddependencies_2escope')
/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitRemoved ('libcontainer-19247-systemd-test-default-dependencies.scope', objectpath '/org/freedesktop/systemd1/unit/libcontainer_2d19247_2dsystemd_2dtest_2ddefault_2ddependencies_2escope')
/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitNew ('libcontainer-19247-systemd-test-default-dependencies.scope', objectpath '/org/freedesktop/systemd1/unit/libcontainer_2d19247_2dsystemd_2dtest_2ddefault_2ddependencies_2escope')
/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitRemoved ('libcontainer-19247-systemd-test-default-dependencies.scope', objectpath '/org/freedesktop/systemd1/unit/libcontainer_2d19247_2dsystemd_2dtest_2ddefault_2ddependencies_2escope')
/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitNew ('libcontainer-19247-systemd-test-default-dependencies.scope', objectpath '/org/freedesktop/systemd1/unit/libcontainer_2d19247_2dsystemd_2dtest_2ddefault_2ddependencies_2escope')
/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitRemoved ('libcontainer-19247-systemd-test-default-dependencies.scope', objectpath '/org/freedesktop/systemd1/unit/libcontainer_2d19247_2dsystemd_2dtest_2ddefault_2ddependencies_2escope')
/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitNew ('libcontainer-19247-systemd-test-default-dependencies.scope', objectpath '/org/freedesktop/systemd1/unit/libcontainer_2d19247_2dsystemd_2dtest_2ddefault_2ddependencies_2escope')
/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitRemoved ('libcontainer-19247-systemd-test-default-dependencies.scope', objectpath '/org/freedesktop/systemd1/unit/libcontainer_2d19247_2dsystemd_2dtest_2ddefault_2ddependencies_2escope')
/org/freedesktop/systemd1: org.freedesktop.systemd1.Manager.UnitNew ('libcontainer_19247_systemd_test_default.slice', objectpath '/org/freedesktop/systemd1/unit/libcontainer_5f19247_5fsystemd_5ftest_5fdefault_2eslice')
```

分析问题发生前后的系统日志，runC 在重复的跑一个 libcontainer_%d_systemd_test_default.slice 测试，这个测试非常频繁，但是当问题发生的时候，这个测试就停止了。所以直觉告诉我，这个问题，可能和这个测试，有很大的关系。



```
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Created slice libcontainer_21185_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Starting libcontainer_21185_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Removed slice libcontainer_21185_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Stopping libcontainer_21185_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Scope libcontainer-21190-systemd-test-default-dependencies.scope has no PIDs. Refusing.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Scope libcontainer-21190-systemd-test-default-dependencies.scope has no PIDs. Refusing.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Created slice libcontainer_21190_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Starting libcontainer_21190_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Removed slice libcontainer_21190_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Stopping libcontainer_21190_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Scope libcontainer-21195-systemd-test-default-dependencies.scope has no PIDs. Refusing.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Scope libcontainer-21195-systemd-test-default-dependencies.scope has no PIDs. Refusing.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Created slice libcontainer_21195_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Starting libcontainer_21195_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Removed slice libcontainer_21195_systemd_test_default.slice.
Sep 29 20:42:16 iz2ze57isplalillyvapgkz systemd[1]: Stopping libcontainer_21195_systemd_test_default.slice.
Sep 29 20:43:05 iz2ze57isplalillyvapgkz kubelet[8031]: I0929 20:43:05.208367    8031 logs.go:49] http: TLS handshake error from 127.0.0.1:45830: EOF
Sep 29 20:43:53 iz2ze57isplalillyvapgkz etcd[825]: store.index: compact 21984245
Sep 29 20:43:53 iz2ze57isplalillyvapgkz etcd[825]: finished scheduled compaction at 21984245 (took 1.705598ms)
Sep 29 20:44:05 iz2ze57isplalillyvapgkz kubelet[8031]: I0929 20:44:05.358462    8031 logs.go:49] http: TLS handshake error from 127.0.0.1:46102: EOF
Sep 29 20:45:05 iz2ze57isplalillyvapgkz kubelet[8031]: I0929 20:45:05.510042    8031 logs.go:49] http: TLS handshake error from 127.0.0.1:46374: EOF
Sep 29 20:46:05 iz2ze57isplalillyvapgkz kubelet[8031]: I0929 20:46:05.660193    8031 logs.go:49] http: TLS handshake error from 127.0.0.1:46650: EOF
Sep 29 20:47:05 iz2ze57isplalillyvapgkz kubelet[8031]: I0929 20:47:05.810903    8031 logs.go:49] http: TLS handshake error from 127.0.0.1:46928: EOF
Sep 29 20:48:05 iz2ze57isplalillyvapgkz kubelet[8031]: I0929 20:48:05.961405    8031 logs.go:49] http: TLS handshake error from 127.0.0.1:47200: EOF
Sep 29 20:48:53 iz2ze57isplalillyvapgkz etcd[825]: store.index: compact 21985079
Sep 29 20:48:53 iz2ze57isplalillyvapgkz etcd[825]: finished scheduled compaction at 21985079 (took 1.697561ms)
Sep 29 20:49:06 iz2ze57isplalillyvapgkz kubelet[8031]: I0929 20:49:06.112241    8031 logs.go:49] http: TLS handshake error from 127.0.0.1:47476: EOF
Sep 29 20:50:06 iz2ze57isplalillyvapgkz kubelet[8031]: I0929 20:50:06.262958    8031 logs.go:49] http: TLS handshake error from 127.0.0.1:47750: EOF
```

另外，我使用 systemd-analyze 命令，打开了 systemd 的调试日志，发现 systemd 有 Operation not supported 的报错。



根据以上零散的知识，只能做出一个大概的结论：org.freedesktop.systemd1 这个 bus 在经过大量 Unit 创建删除之后，没有了响应。而这些频繁的 Unit 创建删除测试，是 runC 某一个 checkin 改写了 UseSystemd 这个函数，而这个函数被用来测试，systemd 的某些功能是否可用。UseSystemd 这个函数在很多地方被调用，比如创建容器，或者查看容器性能等操作。

代码分析

这个问题在线上所有 Kubernetes 集群中，发生的频率大概是一个月两例。问题一直在发生，且只能在问题发生之后，通过重启 systemd 来处理，这风险极大。

我们分别给 systemd 和 runC 社区提交了 bug，但是一个很现实的问题是，他们并没有像阿里云这样的线上环境，他们重现这个问题的概率几乎是零，所以这个问题没有办法指望社区来解决。硬骨头还得我们自己啃。

在上一节最后，我们看到了，问题出现的时候，systemd 会输出一些 Operation not supported 报错。这个报错看起来和问题本身风马牛不相及，但是直觉告诉我，这，或许是离问题最近的一个地方，所以我决定，先搞清楚这个报错因何而来。

Systemd 代码量比较大，而报这个错误的地方也比较多。通过大量的代码分析（这里略去一千字），我发现有几处比较可疑地方，有了这些可疑的地方，接下来需要

做的事情，就是等待。在等待了三周以后，终于有线上集群，再次重现了这个问题。

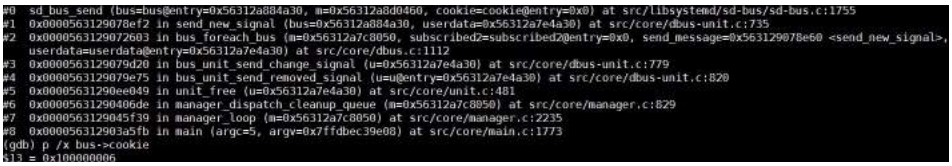
Live Debugging

在征求客户同意之后，下载 systemd 调试符号，挂载 gdb 到 systemd 上，在可疑的函数下断点，continue 继续执行。经过多次验证，发现 systemd 最终踩到了 sd_bus_message_seal 这个函数里的 EOPNOTSUPP 报错。

```
_public_ int sd_bus_message_seal(sd_bus_message *m, uint64_t cookie, uint64_t timeout_usec) {
        struct bus_body_part *part;
        size_t a;
        unsigned i;
        int r;

        assert_return(m, -EINVAL);

        if (m->sealed)
                return -EPERM;

        if (m->n_containers > 0)
                return -EBADMSG;

        if (m->poisoned)
                return -ESTALE;

        if (cookie > 0xffffffffULL &&
            !BUS_MESSAGE_IS_GVARIANT(m))
                return -EOPNOTSUPP;
```

这个报错背后的道理是，systemd 使用了一个变量 cookie，来追踪自己处理的所有 dbus message。每次在在加封一个新的消息的时候，systemd 都会先把 cookie 这个值加一，然后再把这个 cookie 值复制给这个新的 message。

我们使用 gdb 打印出 dbus->cookie 这个值，可以很清楚看到，这个值超过了 0xffffffff。所以看起来，这个问题是 systemd 在加封过大量 message 之后，cookie 这个值 32 位溢出，新的消息不能被加封导致的。



另外，在一个正常的系统上，使用 gdb 把 bus->cookie 这个值改到接近 0xffffffff，然后观察到，问题在 cookie 溢出的时候立刻出现，则证明了我们的结论。

怎么判断集群节点 NotReady 是这个问题导致的

首先我们需要在有问题的节点上安装 gdb 和 systemd debuginfo，然后用命令 gdb /usr/lib/systemd/systemd 1 把 gdb attach 到 systemd，在函数 sd_bus_send 设置断点，然后继续执行。等 systemd 踩到断点之后，用 p /x bus->cookie 查看对应的 cookie 值，如果此值超过了 0xffffffff，那么 cookie 就溢出了，则必然导致节点 NotReady 的问题。确认完之后，可以使用 quit 来 detach 调试器。

问题修复

这个问题的修复，并没有那么直截了当。原因之一，是 systemd 使用了同一个 cookie 变量，来兼容 dbus1 和 dbus2。对于 dbus1 来说，cookie 是 32 位的，这个值在经过 systemd 三五个月频繁创建删除 Unit 之后，是肯定会溢出的；而 dbus2 的 cookie 是 64 位的，可能到了时间的尽头，它也不会溢出。

另外一个原因是，我们并不能简单的让 cookie 折返，来解决溢出问题。因为这有可能导致 systemd 使用同一个 cookie 来加封不同的消息，这样的结果将是灾难性的。

最终的修复方法是，使用 32 位 cookie 来同样处理 dbus1 和 dbus2 两种情形。同时在 cookie 达到 0xffffffff 的之后下一个 cookie 返回 0x80000000，用最高位来标记 cookie 已经处于溢出状态。检查到 cookie 处于这种状态时，我们需要检查是否下一个 cookie 正在被其他 message 使用，来避免 cookie 冲突。

后记

这个问题根本原因肯定在 systemd，但是 runC 的函数 UseSystemd 使用不那么美丽的方法，去测试 systemd 的功能，而这个函数在整个容器生命周期管理过程中，被频繁的触发，让这个低概率问题的发生成为了可能。systemd 的修复已经被红帽接受，预期不久的将来，我们可以通过升级 systemd，从根本上解决这个问题。

https://github.com/lnykryn/systemd-rhel/pull/322

节点下线姊妹篇

简介：之前分享过一例集群节点 NotReady 的问题。在那个问题中，我们的排查路劲，从 K8S 集群到容器运行时，再到 sdbus 和 systemd，不可谓不复杂。那个问题目前已经在 systemd 中做了修复，所以基本上能看到那个问题的几率是越来越低了。

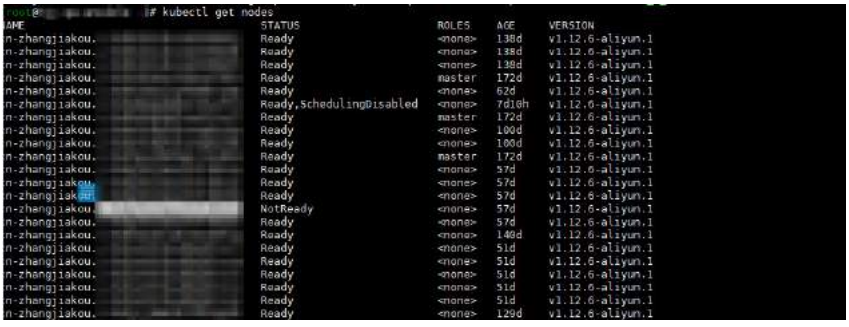
之前分享过一例集群节点 NotReady 的问题。在那个问题中，我们的排查路劲，从 K8S 集群到容器运行时，再到 sdbus 和 systemd，不可谓不复杂。那个问题目前已经在 systemd 中做了修复，所以基本上能看到那个问题的几率是越来越低了。

但是，集群节点就绪问题还是有的，然而原因却有所不同。

今天这篇文章，跟大家分享另外一例集群节点 NotReady 的问题。这个问题和之前那个问题相比，排查路劲完全不同。作为姊妹篇分享给大家。

问题现象

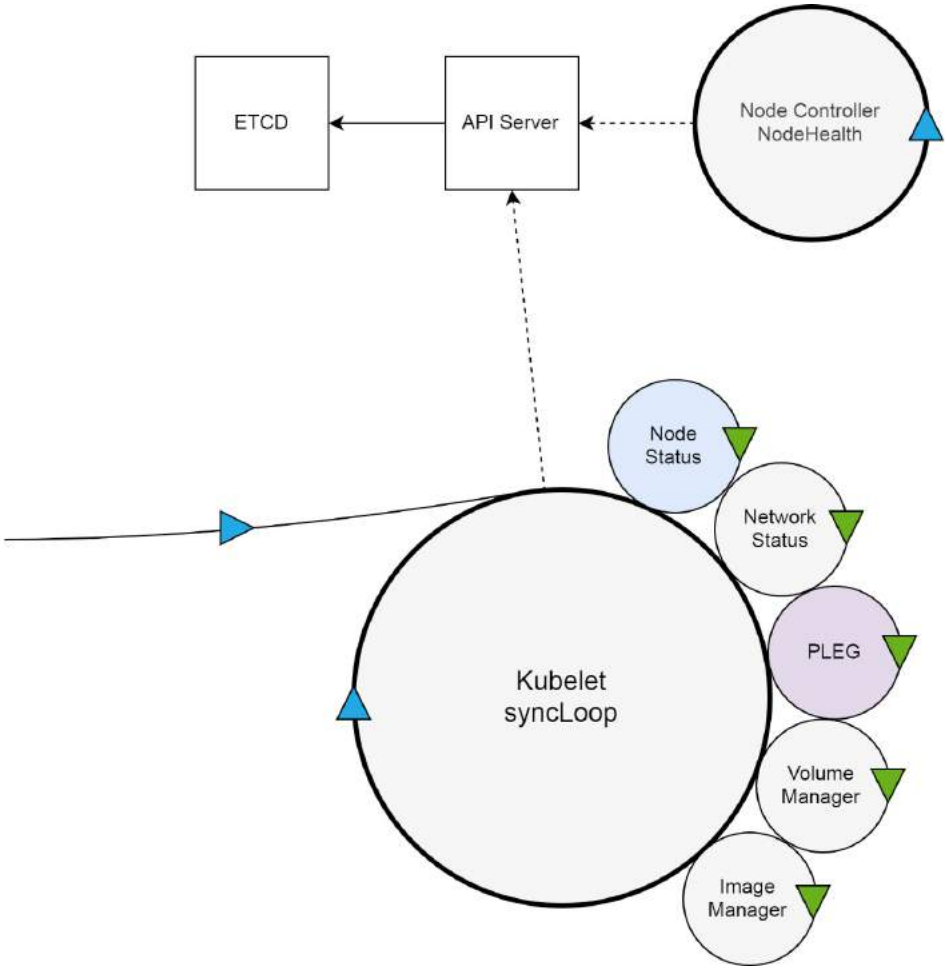
这个问题的现象，也是集群节点会变成 NotReady 状态。问题可以通过重启节点暂时解决，但是在经过大概 20 天左右之后，问题会再次出现。



问题出现之后，如果我们重启节点上 kubelet，则节点会变成 Ready 状态，但这种状态只会持续三分钟。这是一个特别的情况。

大逻辑

在具体分析这个问题之前，我们先来看一下集群节点就绪状态背后的大逻辑。K8S 集群中，与节点就绪状态有关的组件，主要有四个，分别是集群的核心数据库 etcd，集群的入口 API Server，节点控制器以及驻守在集群节点上，直接管理节点的 kubelet。



一方面，kubelet 扮演的是集群控制器的角色，它定期从 API Server 获取 Pod 等相关资源的信息，并依照这些信息，控制运行在节点上 Pod 的执行；另外一方面，kubelet 作为节点状况的监视器，它获取节点信息，并以集群客户端的角色，把这些状况同步到 API Server。

在这个问题中，kubelet 扮演的是第二种角色。

Kubelet 会使用上图中的 NodeStatus 机制，定期检查集群节点状况，并把节点状况同步到 API Server。而 NodeStatus 判断节点就绪状况的一个主要依据，就是 PLEG。

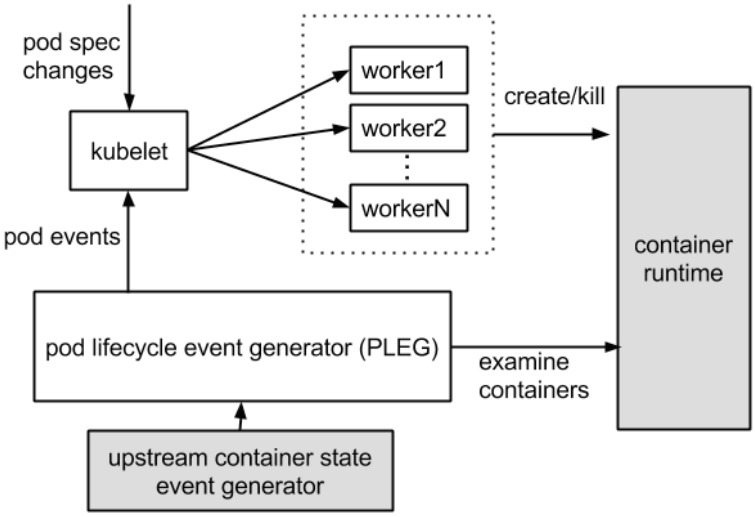
PLEG 是 Pod Lifecycle Events Generator 的缩写，基本上它的执行逻辑，是定期检查节点上 Pod 运行情况，如果发现感兴趣的变化，PLEG 就会把这种变化包装成 Event 发送给 Kubelet 的主同步机制 syncLoop 去处理。但是，在 PLEG 的 Pod 检查机制不能定期执行的时候，NodeStatus 机制就会认为，这个节点的状况是不对的，从而把这种状况同步到 API Server。

而最终把 kubelet 上报的节点状况，落实到节点状态的是节点控制这个组件。这里我故意区分了 kubelet 上报的节点状况，和节点的最终状态。因为前者，其实是我们 describe node 时看到的 Condition，而后者是真正节点列表里的 NotReady 状态。

```
Conditions:
  Type                 Status  LastHeartbeatTime                 LastTransitionTime                Reason                       Message
  ----                 ------  -----------------                 ------------------                ------                       -------
  NetworkUnavailable   False   Mon, 09 Sep 2019 00:04:30 +0800   Mon, 09 Sep 2019 00:04:30 +0800   RouteCreated                 RouteController created a route
  OutOfDisk            False   Mon, 09 Sep 2019 00:04:57 +0800   Wed, 21 Aug 2019 22:03:10 +0800   KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure       False   Mon, 09 Sep 2019 00:04:57 +0800   Wed, 21 Aug 2019 22:03:10 +0800   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure         False   Mon, 09 Sep 2019 00:04:57 +0800   Wed, 21 Aug 2019 22:03:10 +0800   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure          False   Mon, 09 Sep 2019 00:04:57 +0800   Tue, 13 Aug 2019 11:44:49 +0800   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready                True    Mon, 09 Sep 2019 00:04:57 +0800   Wed, 21 Aug 2019 22:03:10 +0800   KubeletReady                 kubelet is posting ready status
```

就绪三分钟

在问题发生之后，我们重启 kubelet，节点三分钟之后才会变成 NotReady 状态。这个现象是问题的一个关键切入点。

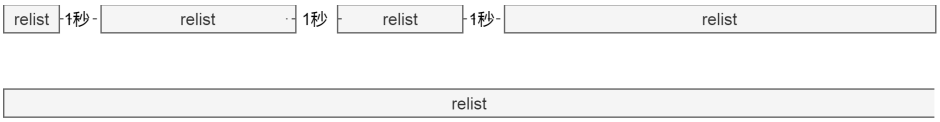


在解释它之前，请大家看一下官方这张 PLEG 示意图。这个图片主要展示了两个过程。一方面，kubelet 作为集群控制器，从 API Server 处获取 pod spec changes，然后通过创建 worker 线程来创建或结束掉 pod；另外一方面，PLEG 定期检查容器状态，然后把状态，以事件的形式反馈给 kubelet。

在这里，PLEG 有两个关键的时间参数，一个是检查的执行间隔，另外一个是检查的超时时间。以默认情况为准，PLEG 检查会间隔一秒，换句话说，每一次检查过程执行之后，PLEG 会等待一秒钟，然后进行下一次检查；而每一次检查的超时时间是三分钟，如果一次 PLEG 检查操作不能在三分钟内完成，那么这个状况，会被上一节提到的 NodeStatus 机制，当做集群节点 NotReady 的凭据，同步给 API Server。

而我们之所以观察到节点会在重启 kubelet 之后就绪三分钟，是因为 kubelet 重启之后，第一次 PLEG 检查操作就没有顺利结束。节点就绪状态，直到三分钟超时之后，才被同步到集群。

如下图，上边一行表示正常情况下 PLEG 的执行流程，下边一行则表示有问题的情况。relist 是检查的主函数。



止步不前的 PLEG

了解了原理，我们来看一下 PLEG 的日志。日志基本上可以分为两部分，其中 skipping pod synchronization 这部分是 kubelet 同步函数 syncLoop 输出的，说明它跳过了一次 pod 同步；而剩余 PLEG is not healthy: pleg was last seen active ago; threshold is 3m0s，则很清楚的展现了，上一节提到的 relist 超时三分钟的问题。

```
17:08:22.299597 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m0.000091019s ago; threshold is 3m0s]
17:08:22.399758 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m0.100259802s ago; threshold is 3m0s]
17:08:22.599931 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m0.300436887s ago; threshold is 3m0s]
17:08:23.000087 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m0.700575691s ago; threshold is 3m0s]
17:08:23.800258 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m1.500754856s ago; threshold is 3m0s]
17:08:25.400439 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m3.100936232s ago; threshold is 3m0s]
17:08:28.600599 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m6.301098811s ago; threshold is 3m0s]
17:08:33.600812 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m11.30128783s ago; threshold is 3m0s]
17:08:38.600983 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m16.301473637s ago; threshold is 3m0s]
17:08:43.601157 kubelet skipping pod synchronization - [PLEG is not healthy:
```

```
pleg was last seen active
3m21.301651575s ago; threshold is 3m0s]
17:08:48.601331 kubelet skipping pod synchronization - [PLEG is not healthy:
pleg was last seen active
3m26.301826001s ago; threshold is 3m0s]
```

能直接看到 relist 函数执行情况的，是 kubelet 的调用栈。我们只要向 kubelet 进程发送 SIGABRT 信号，golang 运行时就会帮我们输出 kubelet 进程的所有调用栈。需要注意的是，这个操作会杀死 kubelet 进程。但是因为这个问题中，重启 kubelet 并不会破坏重现环境，所以影响不大。

以下调用栈是 PLEG relist 函数的调用栈。从下往上，我们可以看到，relist 等在通过 grpc 获取 PodSandboxStatus。

```
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc/transport.(*Stream).
Header()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.recvResponse()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.invoke()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.Invoke()
kubelet: k8s.io/kubernetes/pkg/kubelet/apis/cri/runtime/v1alpha2.
(*runtimeServiceClient).
PodSandboxStatus()
kubelet: k8s.io/kubernetes/pkg/kubelet/remote.(*RemoteRuntimeService).
PodSandboxStatus()
kubelet: k8s.io/kubernetes/pkg/kubelet/kuberuntime.
instrumentedRuntimeService.
PodSandboxStatus()
kubelet: k8s.io/kubernetes/pkg/kubelet/kuberuntime.
(*kubeGenericRuntimeManager).GetPodStatus()
kubelet: k8s.io/kubernetes/pkg/kubelet/pleg.(*GenericPLEG).updateCache()
kubelet: k8s.io/kubernetes/pkg/kubelet/pleg.(*GenericPLEG).relist()
kubelet: k8s.io/kubernetes/pkg/kubelet/pleg.(*GenericPLEG).(k8s.io/
kubernetes/pkg/kubelet/pleg.
relist)-fm()
kubelet: k8s.io/kubernetes/vendor/k8s.io/apimachinery/pkg/util/wait.
JitterUntil.func1(0xc420309260)
kubelet: k8s.io/kubernetes/vendor/k8s.io/apimachinery/pkg/util/wait.
JitterUntil()
kubelet: k8s.io/kubernetes/vendor/k8s.io/apimachinery/pkg/util/wait.Until()
```

使用 PodSandboxStatus 搜索 kubelet 调用栈，很容易找到下边这个线程，此线程是真正查询 Sandbox 状态的线程，从下往上看，我们会发现这个线程在 Plugin

Manager 里尝试去拿一个 Mutex。

```
kubelet: sync.runtime_SemacquireMutex()kubelet: sync.(*Mutex).Lock()
kubelet: k8s.io/kubernetes/pkg/kubelet/dockershim/network.(*PluginManager).
GetPodNetworkStatus()
kubelet: k8s.io/kubernetes/pkg/kubelet/dockershim.(*dockerService).
getIPFromPlugin()
kubelet: k8s.io/kubernetes/pkg/kubelet/dockershim.(*dockerService).getIP()
kubelet: k8s.io/kubernetes/pkg/kubelet/dockershim.(*dockerService).
PodSandboxStatus()
kubelet: k8s.io/kubernetes/pkg/kubelet/apis/cri/runtime/v1alpha2._
RuntimeService_
PodSandboxStatus_Handler()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.(*Server).
processUnaryRPC()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.(*Server).
handleStream()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.(*Server).
serveStreams.func1.1()
kubelet: created by k8s.io/kubernetes/vendor/google.golang.org/grpc.
(*Server).serveStreams.func1
```

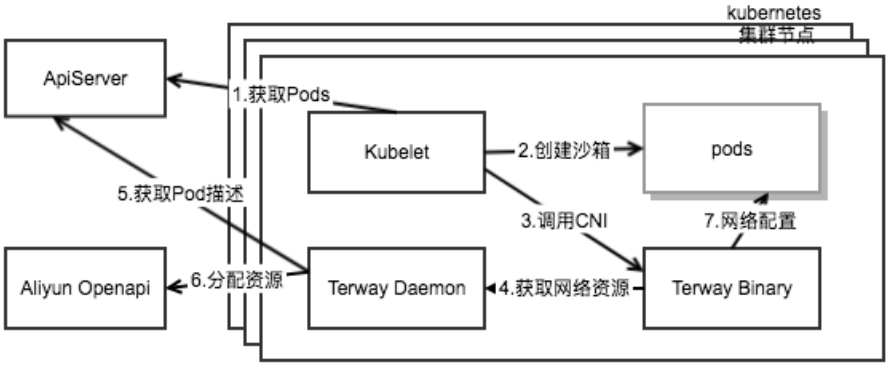
而这个 Mutex 只有在 Plugin Manager 里边有用到，所以我们查看所有 Plugin Manager 相关的调用栈。线程中一部分在等 Mutex，而剩余的都是在等 Terway cni plugin。

```
kubelet: syscall.Syscall6()kubelet: os.(*Process).blockUntilWaitable()
kubelet: os.(*Process).wait()kubelet: os.(*Process).Wait()
kubelet: os/exec.(*Cmd).Wait()kubelet: os/exec.(*Cmd).Run()
kubelet: k8s.io/kubernetes/vendor/github.com/containernetworking/cni/pkg/
invoke.(*RawExec).
ExecPlugin()
kubelet: k8s.io/kubernetes/vendor/github.com/containernetworking/cni/pkg/
invoke.(*PluginExec).
WithResult()
kubelet: k8s.io/kubernetes/vendor/github.com/containernetworking/cni/pkg/
invoke.
ExecPluginWithResult()
kubelet: k8s.io/kubernetes/vendor/github.com/containernetworking/cni/libcni.
(*CNIConfig).
AddNetworkList()
kubelet: k8s.io/kubernetes/pkg/kubelet/dockershim/network/cni.
(*cniNetworkPlugin).
addToNetwork()
kubelet: k8s.io/kubernetes/pkg/kubelet/dockershim/network/cni.
```

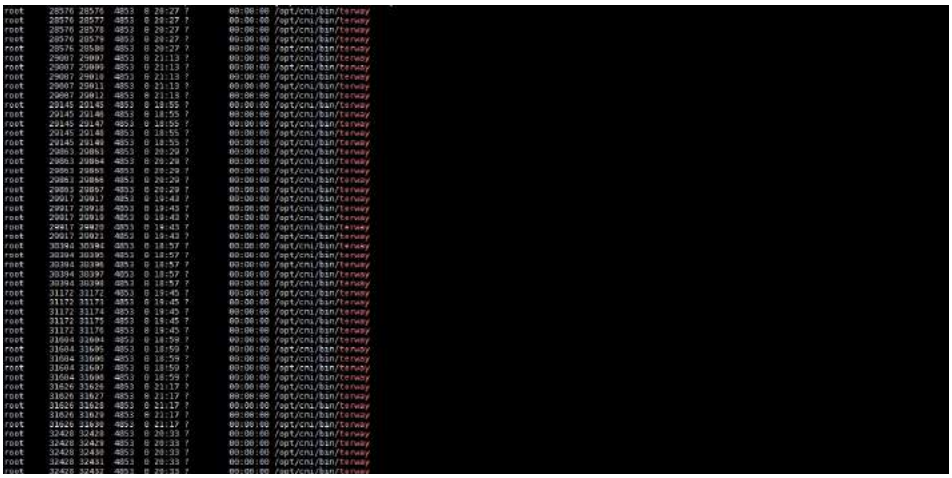
```
(*cniNetworkPlugin).SetUpPod()
kubelet: k8s.io/kubernetes/pkg/kubelet/dockershim/network.(*PluginManager).
SetUpPod()
kubelet: k8s.io/kubernetes/pkg/kubelet/dockershim.(*dockerService).
RunPodSandbox()
kubelet: k8s.io/kubernetes/pkg/kubelet/apis/cri/runtime/v1alpha2._
RuntimeService_
RunPodSandbox_Handler()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.(*Server).
processUnaryRPC()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.(*Server).
handleStream()
kubelet: k8s.io/kubernetes/vendor/google.golang.org/grpc.(*Server).
serveStreams.func1.1()
```

无响应的 Terwayd

在进一步解释这个问题之前，我们需要区分下 Terway 和 Terwayd。本质上来说，Terway 和 Terwayd 是客户端服务器的关系，这跟 flannel 和 flanneld 之间的关系是一样的。Terway 是按照 kubelet 的定义，实现了 cni 接口的插件。



而在上一节最后，我们看到的问题，是 kubelet 调用 CNI terway 去配置 pod 网络的时候，Terway 长时间无响应。正常情况下这个操作应该是秒级的，非常快速。而出问题的时候，Terway 没有正常完成任务，因而我们在集群节点上看到大量 terway 进程堆积。



同样的，我们可以发送 SIGABRT 给这些 terway 插件进程，来打印出进程的调用栈。下边是其中一个 terway 的调用栈。这个线程在执行 cmdDel 函数，其作用是删除一个 pod 网络相关配置。

```
kubelet: net/rpc.(*Client).Call()
kubelet: main.rpcCall()kubelet: main.cmdDel()
kubelet: github.com/AliyunContainerService/terway/vendor/github.com/
containernetworking/cni/
pkg/skel.(*dispatcher).checkVersionAndCall()
kubelet: github.com/AliyunContainerService/terway/vendor/github.com/
containernetworking/cni/
pkg/skel.(*dispatcher).pluginMain()
kubelet: github.com/AliyunContainerService/terway/vendor/github.com/
containernetworking/cni/
pkg/skel.PluginMainWithError()
kubelet: github.com/AliyunContainerService/terway/vendor/github.com/
containernetworking/cni/
pkg/skel.PluginMain()
```

以上线程通过 rpc 调用 terwayd，来真正的移除 pod 网络。所以我们需要进一步排查 terwayd 的调用栈来进一步定位此问题。Terwayd 作为 Terway 的服务器端，其接受 Terway 的远程调用，并替 Terway 完成其 cmdAdd 或者 cmdDel 来创建或者移除 pod 网络配置。

我们在上边的截图里可以看到，集群节点上有成千 Terway 进程，他们都在等待 Terwayd，所以实际上 Terwayd 里，也有成千的线程在处理 Terway 的请求。

使用下边的命令，可以在不重启 Terwayd 的情况下，输出调用栈。

```
curl  --unix-socket /var/run/eni/eni.socket 'http:/debug/pprof/
goroutine?debug=2'
```

因为 Terwayd 的调用栈非常复杂，而且几乎所有的线程都在等锁，直接去分析锁的等待持有关系比较复杂。这个时候我们可以使用“时间大法”，即假设最早进入等待状态的线程，大概率是持有锁的线程。

经过调用栈和代码分析，我们发现下边这个是等待时间最长（1595 分钟），且拿了锁的线程。而这个锁会 block 所有创建或者销毁 pod 网络的线程。

```
goroutine 67570 [syscall, 1595 minutes, locked to thread]:
syscall.Syscall6()
github.com/AliyunContainerService/terway/vendor/golang.org/x/sys/unix.
recvfrom()
github.com/AliyunContainerService/terway/vendor/golang.org/x/sys/unix.
Recvfrom()
github.com/AliyunContainerService/terway/vendor/github.com/vishvananda/
netlink/
nl.(*NetlinkSocket).Receive()
github.com/AliyunContainerService/terway/vendor/github.com/vishvananda/
netlink/
nl.(*NetlinkRequest).Execute()
github.com/AliyunContainerService/terway/vendor/github.com/vishvananda/
netlink.(*Handle).
LinkSetNsFd()
github.com/AliyunContainerService/terway/vendor/github.com/vishvananda/
netlink.LinkSetNsFd()
github.com/AliyunContainerService/terway/daemon.SetupVethPair()github.com/
AliyunContainerService/terway/daemon.setupContainerVeth.func1()
github.com/AliyunContainerService/terway/vendor/github.com/
containernetworking/plugins/pkg/
ns.(*netNS).Do.func1()
github.com/AliyunContainerService/terway/vendor/github.com/
containernetworking/plugins/pkg/
ns.(*netNS).Do.func2()
```

原因深入分析前一个线程的调用栈，我们可以确定三件事情。第一，Terwayd 使用了 netlink 这个库来管理节点上的虚拟网卡，IP 地址及路由等资源，且 netlink 实现了类似 iproute2 的功能；第二，netlink 使用 socket 直接和内核通信；第三，以上线程等在 recvfrom 系统调用上。

这样的情况下，我们需要去查看这个线程的内核调用栈，才能进一步确认这个线程等待的原因。因为从 goroutine 线程号比较不容易找到这个线程的系统线程 id，这里我们通过抓取系统的 core dump 来找出上边线程的内核调用栈。

在内核调用栈中，搜索 recvfrom，定位到下边这个线程。基本上从下边的调用栈上，我们只能确定，此线程等在 recvfrom 函数上。

```
PID: 19246  TASK: ffff880951f70fd0  CPU: 16  COMMAND: "terwayd"
#0 [ffff880826267a40] __schedule at ffffffff816a8f65
#1 [ffff880826267aa8] schedule at ffffffff816a94e9
#2 [ffff880826267ab8] schedule_timeout at ffffffff816a6ff9
#3 [ffff880826267b68] __skb_wait_for_more_packets at ffffffff81578f80
#4 [ffff880826267bd0] __skb_recv_datagram at ffffffff8157935f
#5 [ffff880826267c38] skb_recv_datagram at ffffffff81579403
#6 [ffff880826267c58] netlink_recvmsg at ffffffff815bb312
#7 [ffff880826267ce8] sock_recvmsg at ffffffff8156a88f
#8 [ffff880826267e58] SYSC_recvfrom at ffffffff8156aa08
#9 [ffff880826267f70] sys_recvfrom at ffffffff8156b2fe
#10 [ffff880826267f80] tracesys at ffffffff816b5212
(via system_call)
```

这个问题进一步深入排查，是比较困难的，这显然是一个内核问题，或者内核相关的问题。我们翻遍了整个内核 core，检查了所有的线程调用栈，看不到其他可能与这个问题相关联的线程。

修复

这个问题的修复基于一个假设，就是 netlink 并不是 100% 可靠的。netlink 可以响应很慢，甚至完全没有响应。所以我们可以给 netlink 操作增加超时，从而保证就算某一次 netlink 调用不能完成的情况下，terwayd 也不会被阻塞。

总结

在节点就绪状态这种场景下，kubelet 实际上实现了节点的心跳机制。kubelet 会定期把节点相关的各种状态，包括内存，PID，磁盘，当然包括这个文章中关注的就绪状态等，同步到集群管控。kubelet 在监控或者管理集群节点的过程中，使用了各种插件来直接操作节点资源。这包括网络，磁盘，甚至容器运行时等插件，这些插件的状况，会直接应用 kubelet 甚至节点的状态。

我们为什么会删除不了集群的命名空间?

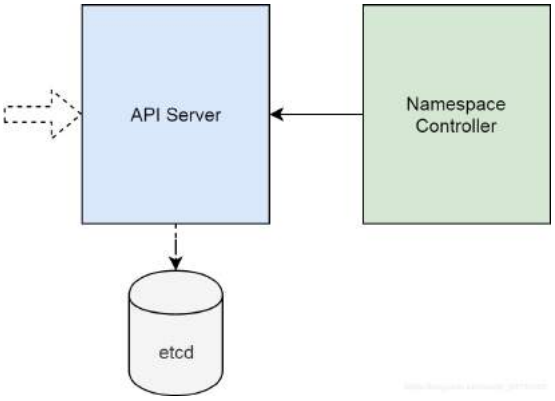
阿里云售后技术团队的同学，每天都在处理各式各样千奇百怪的线上问题。常见的有，网络连接失败，服务器宕机，性能不达标，请求响应慢等。但如果要评选，什么问题看起来微不足道事实上却足以让人绞尽脑汁，我相信答案肯定是“删不掉”的问题。比如文件删不掉，进程结束不掉，驱动卸载不了等。

这样的问题就像冰山，影藏在它们背后的复杂逻辑，往往超过我们的预想。

背景

今天我们讨论的这个问题，跟 K8S 集群的命名空间有关。命名空间是 K8S 集群资源的“收纳”机制。我们可以把相关的资源，“收纳”到同一个命名空间里，以避免不相关资源之间不必要的影响。

命名空间本身也是一种资源。通过集群 API Server 入口，我们可以新建命名空间，而对于不再使用的命名空间，我们需要清理掉。命名空间的 Controller 会通过 API Server，监视集群中命名空间的变化，然后根据变化来执行预先定义的动作。



有时候，我们会遇到下图中的问题，即命名空间的状态被标记成了“Terminating”，但却没有办法被完全删除。

```
[root@[redacted] ~]# kubectl get ns
NAME           STATUS        AGE
catalog        Active        18d
default        Active        21d
dev            Terminating   18m
istio-system   Active        9d
kube-public    Active        21d
kube-system    Active        21d
space          Terminating   21m
```

从集群入口开始

因为删除操作是通过集群 API Server 来执行的，所以我们要分析 API Server 的行为。跟大多数集群组件类似，API Server 提供了不同级别的日志输出。为了理解 API Server 的行为，我们将日志级别调整到最高级。然后，通过创建删除 tobedeletedb 这个命名空间来重现问题。

但可惜的是，API Server 并没有输出太多和这个问题有关的日志。

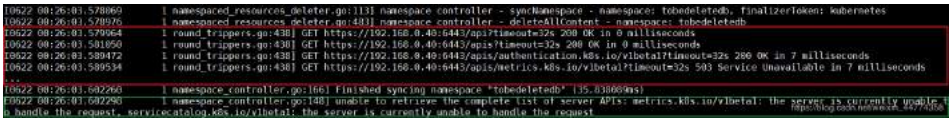
相关的日志，可以分为两部分。一部分是命名空间被删除的记录，记录显示客户端工具是 kubectl，以及发起操作的源 IP 地址是 192.168.0.41，这符合预期；另外一部分是 Kube Controller Manager 在重复的获取这个命名空间的信息。

```
I0622 00:25:58.559261       1 handler.go:153] kube-aggregator: DELETE "/api/v1/namespaces/tobedeletedb" satisfied by nonGoRestful
I0622 00:25:58.559298       1 pathrecorder.go:247] kube-aggregator: "/api/v1/namespaces/tobedeletedb" satisfied by prefix /api/
I0622 00:25:58.559311       1 handler.go:143] kube-apiserver: DELETE "/api/v1/namespaces/tobedeletedb" satisfied by gorestful with webservice /api/v1
I0622 00:25:58.565611       1 wrap.go:47] DELETE /api/v1/namespaces/tobedeletedb: (6.599578ms) 200 [kubectl/v1.12.6 (linux/amd64) kubernetes/8cb561c 192.168.0.41:43478]
I0622 00:25:58.568970       1 wrap.go:47] GET /api/v1/namespaces?fieldSelector=metadata.name%3Dtobedeletedb: (2.389303ms) 200 [kubectl/v1.12.6 (linux/amd64) kubernetes/
8cb561c 192.168.0.41:43478]
I0622 00:25:58.570094       1 get.go:245] Starting watch for /api/v1/namespaces, rv=4688342 labels= fields=metadata.name=tobedeletedb timeout=55m56.614289088s
I0622 00:26:03.574913       1 handler.go:153] kube-aggregator: GET "/api/v1/namespaces/tobedeletedb" satisfied by nonGoRestful
I0622 00:26:03.574932       1 pathrecorder.go:247] kube-aggregator: "/api/v1/namespaces/tobedeletedb" satisfied by prefix /api/
I0622 00:26:03.574947       1 handler.go:143] kube-apiserver: GET "/api/v1/namespaces/tobedeletedb" satisfied by gorestful with webservice /api/v1
I0622 00:26:03.577435       1 wrap.go:47] GET /api/v1/namespaces/tobedeletedb: (10.412766ms) 200 [kube-controller-manager/v1.12.6 (linux/amd64) kubernetes/8cb561c/syste
m:serviceaccount:kube-system:namespace-controller 192.168.0.40:52080]
I0622 00:26:03.600310       1 handler.go:153] kube-aggregator: GET "/api/v1/namespaces/tobedeletedb" satisfied by nonGoRestful
I0622 00:26:03.600330       1 pathrecorder.go:247] kube-aggregator: "/api/v1/namespaces/tobedeletedb" satisfied by prefix /api/
I0622 00:26:03.600349       1 handler.go:143] kube-apiserver: GET "/api/v1/namespaces/tobedeletedb" satisfied by gorestful with webservice /api/v1
I0622 00:26:03.613263       1 wrap.go:47] GET /api/v1/namespaces/tobedeletedb: (15.151073ms) 200 [kube-controller-manager/v1.12.6 (linux/amd64) kubernetes/8cb561c/system
:serviceaccount:kube-system:namespace-controller 192.168.0.40:52080]
I0622 00:26:03.639442       1 handler.go:153] kube-aggregator: GET "/api/v1/namespaces/tobedeletedb" satisfied by nonGoRestful
I0622 00:26:03.639464       1 pathrecorder.go:247] kube-aggregator: "/api/v1/namespaces/tobedeletedb" satisfied by prefix /api/
I0622 00:26:03.639473       1 handler.go:143] kube-apiserver: GET "/api/v1/namespaces/tobedeletedb" satisfied by gorestful with webservice /api/v1
I0622 00:26:03.641572       1 wrap.go:47] GET /api/v1/namespaces/tobedeletedb: (2.242154ms) 200 [kube-controller-manager/v1.12.6 (linux/amd64) kubernetes/8cb561c/system
:serviceaccount:kube-system:namespace-controller 192.168.0.40:52080]
```

Kube Controller Manager 实现了集群中大多数的 Controller，它在重复获取 tobedeletedb 的信息，基本上可以判断，是命名空间的 Controller 在获取这个命名空间的信息。

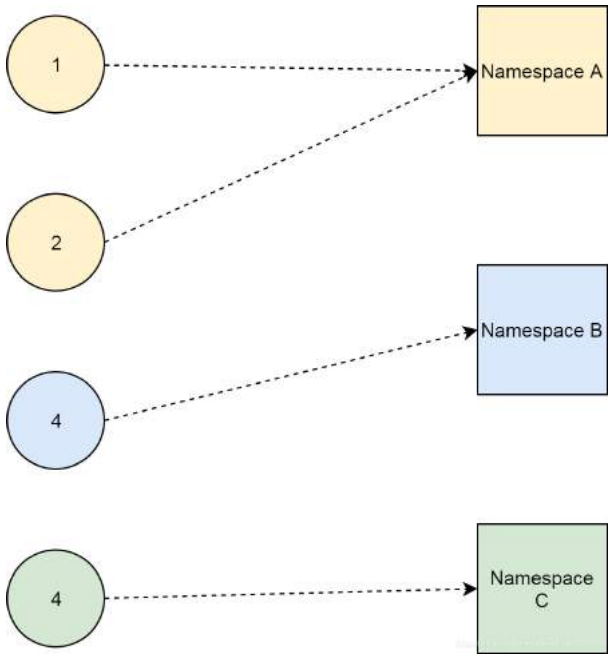
Controller 在做什么？

和上一节类似，我们通过开启 Kube Controller Manager 最高级别日志，来研究这个组件的行为。在 Kube Controller Manager 的日志里，可以看到命名空间的 Controller 在不断地尝试一个失败了的操作，就是清理 tobedeletedb 这个命名空间里“收纳”的资源。



怎么样删除“收纳盒”里的资源

这里我们需要理解一点，就是命名空间作为资源的“收纳盒”，其实是逻辑意义上的概念。它并不像现实中的收纳工具，可以把小的物件收纳其中。命名空间的“收纳”实际上是一种映射关系。



这一点之所以重要，是因为它直接决定了，删除命名空间内部资源的方法。如果是物理意义上的“收纳”，那我们只需要删除“收纳盒”，里边的资源就一并被删除了。而对于逻辑意义上的关系，我们则需要罗列所有资源，并删除那些指向需要删除的命名空间的资源。

API、Group、Version

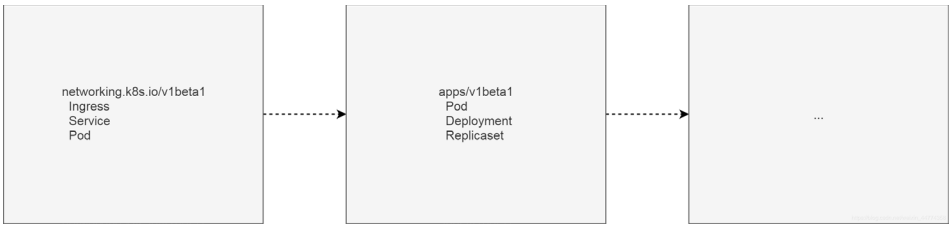
怎么样罗列集群中的所有资源呢，这个问题需要从集群 API 的组织方式说起。K8S 集群的 API 不是铁板一块的，它是用分组和版本来组织的。这样做的好处显而易见，就是不同分组的 API 可以独立的迭代，互不影响。常见的分组如 apps，它有 v1，v1beta1 和 v1beta2 这三个版本。完整的分组 / 版本列表，可以使用 kubectl api-versions 命令看到。

```
[root@i2uf5ivwl1e3r4o4fr2 ~]# kubectl api-versions | grep apps
apps/v1
apps/v1beta1
apps/v1beta2
```

我们创建的每一个资源，都必然属于某一个 API 分组 / 版本。以下边 Ingress 为例，我们指定 Ingress 资源的分组 / 版本为 networking.k8s.io/v1beta1。

```
kind: Ingress
metadata:
  name: test-ingress
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        backend:
          serviceName: test
          servicePort: 80
```

用一个简单的示意图来总结 API 分组和版本。



实际上，集群有很多 API 分组 / 版本，每个 API 分组 / 版本支持特定的资源类型。我们通过 yaml 编排资源时，需要指定资源类型 kind，以及 API 分组 / 版本 apiVersion。而要列出资源，我们需要获取 API 分组 / 版本的列表。

Controller 为什么不能删除命名空间里的资源

理解了 API 分组 / 版本的概念之后，再回头看 Kube Controller Manager 的日志，就会豁然开朗。显然命名空间的 Controller 在尝试获取 API 分组 / 版本列表，当遇到 metrics.k8s.io/v1beta1 的时候，查询失败了。并且查询失败的原因是“the server is currently unable to handle the request”。

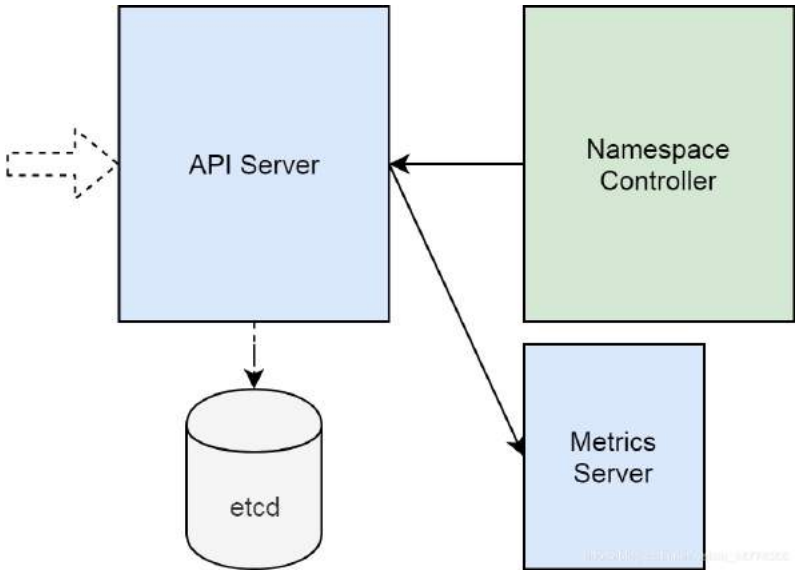
再次回到集群入口

在上一节中，我们发现 Kube Controller Manager 在获取 metrics.k8s.io/v1beta1 这个 API 分组 / 版本的时候失败了。而这个查询请求，显然是发给 API Server 的。所以我们回到 API Server 日志，分析 metrics.k8s.io/v1beta1 相关的记录。在相同的时间点，我们看到 API Server 也报了同样的错误“the server is currently unable to handle the request”。

```
I0622 00:26:02.053417       1 round_trippers.go:438] GET https://localhost:6443/apis/metrics.k8s.io/v1beta1?timeout=32s 503 Service Unavailable in 0 milliseconds
I0622 00:26:02.053497       1 request.go:942] Response Body: service unavailable
E0622 00:26:02.053593       1 memcache.go:134] couldn't get resource list for metrics.k8s.io/v1beta1: the server is currently unable to handle the request
```

显然这里有一个矛盾，就是 API Server 明显在正常工作，为什么在获取 metrics.k8s.io/v1beta1 这个 API 分组版本的时候，会返回 Server 不可用呢？为了

回答这个问题，我们需要理解一下 API Server 的“外挂”机制。



集群 API Server 有扩展自己的机制，开发者可以利用这个机制，来实现 API Server 的“外挂”。这个“外挂”的主要功能，就是实现新的 API 分组 / 版本。API Server 作为代理，会把相应的 API 调用，转发给自己的“外挂”。

以 Metrics Server 为例，它实现了 metrics.k8s.io/v1beta1 这个 API 分组 / 版本。所有针对这个分组 / 版本的调用，都会被转发到 Metrics Server。如下图，Metrics Server 的实现，主要用到一个服务和一个 pod。

```
[root@iZuf6ivvel1xkm3xr4o4frZ ~]# kubectl get pods -n kube-system metrics-server-77f7987bb8-7tgld
NAME                              READY   STATUS    RESTARTS   AGE
metrics-server-77f7987bb8-7tgld   1/1     Running   0          23d
[root@iZuf6ivvel1xkm3xr4o4frZ ~]# kubectl get svc metrics-server -n kube-system
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
metrics-server   ClusterIP   172.21.13.97   <none>        443/TCP   23d
[root@iZuf6ivvel1xkm3xr4o4frZ ~]# kubectl get apiservice v1beta1.metrics.k8s.io
NAME                     SERVICE                      AVAILABLE                      AGE
v1beta1.metrics.k8s.io   kube-system/metrics-server   False (FailedDiscoveryCheck)   23d
```

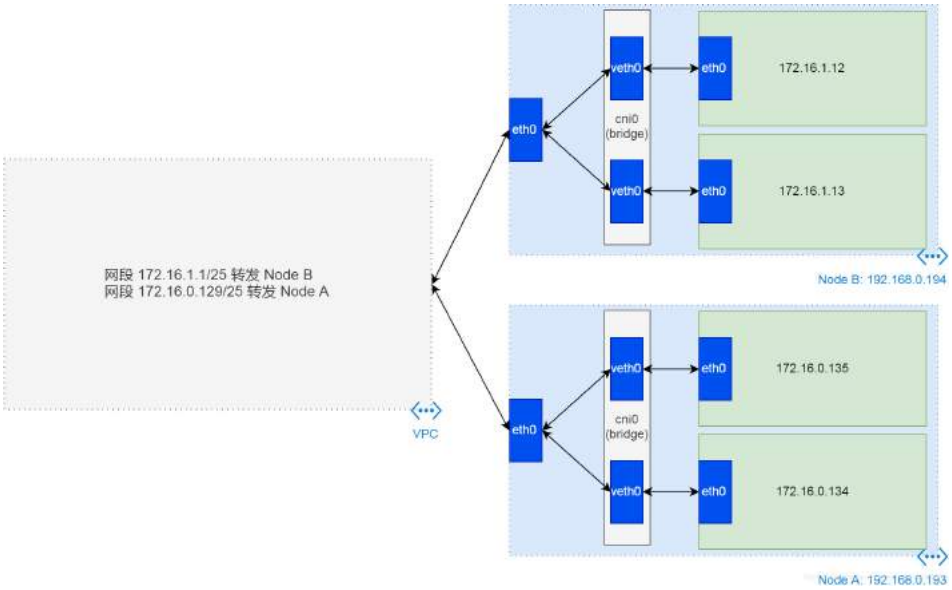
而上图中最后的 apiservice，则是把“外挂”和 API Server 联系起来的机制。下图可以看到这个 apiservice 详细定义。它包括 API 分组 / 版本，以及实现了 Metrics Server 的服务名。有了这些信息，API Server 就能把针对 metrics.k8s.io/

v1beta1 的调用，转发给 Metrics Server。

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  annotations:
  name: v1beta1.metrics.k8s.io
spec:
  group: metrics.k8s.io
  service:
    name: metrics-server
    namespace: kube-system
  version: v1beta1
```


节点与 Pod 之间的通信

经过简单的测试，我们发现，这个问题实际上是 API server 和 metrics server pod 之间的通信问题。在阿里云 K8S 集群环境里，API Server 使用的是主机网络，即 ECS 的网络，而 Metrics Server 使用的是 Pod 网络。这两者之间的通信，依赖于 VPC 路由表的转发。



以上图为例，如果 API Server 运行在 Node A 上，那它的 IP 地址就是 192.168.0.193。假设 Metrics Server 的 IP 是 172.16.1.12，那么从 API Server

到 Metrics Server 的网络连接，必须要通过 VPC 路由表第二条路由规则的转发。

检查集群 VPC 路由表，发现指向 Metrics Server 所在节点的路由表项缺失，所以 API server 和 Metrics Server 之间的通信出了问题。

Route Controller 为什么不工作?

为了维持集群 VPC 路由表项的正确性，阿里云在 Cloud Controller Manager 内部实现了 Route Controller。这个 Controller 在时刻监听着集群节点状态，以及 VPC 路由表状态。当发现路由表项缺失的时候，它会自动把缺失的路由表项填写回去。

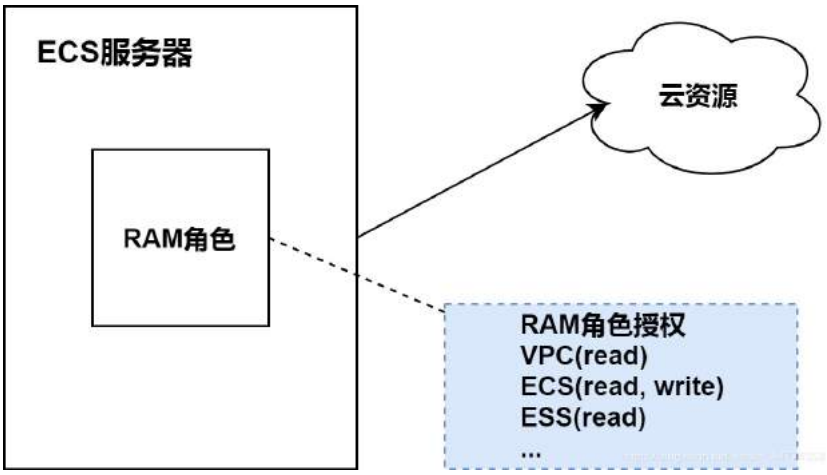
现在的情况，显然和预期不一致，Route Controller 显然没有正常工作。这个可以通过查看 Cloud Controller Manager 日志来确认。在日志中，我们发现，Route Controller 在使用集群 VPC id 去查找 VPC 实例的时候，没有办法获取到这个实例的信息。

但是集群还在，ECS 还在，所以 VPC 不可能不在了。这一点我们可以通过 VPC id 在 VPC 控制台确认。那下边的问题，就是为什么 Cloud Controller Manager 没有办法获取到这个 VPC 的信息呢?

集群节点访问云资源

Cloud Controller Manager 获取 VPC 信息，是通过阿里云开放 API 来实现的。这基本上等于，从云上一台 ECS 内部，去获取一个 VPC 实例的信息，而这需要 ECS 有足够的权限。目前的常规做法是，给 ECS 服务器授予 RAM 角色，同时给对

应的 RAM 角色绑定相应的角色授权。



如果集群组件，以其所在节点的身份，不能获取云资源的信息，那基本上有两种可能性。一是 ECS 没有绑定正确的 RAM 角色；二是 RAM 角色绑定的 RAM 角色授权没有定义正确的授权规则。检查节点的 RAM 角色，以及 RAM 角色所管理的授权，我们发现，针对 vpc 的授权策略被改掉了。

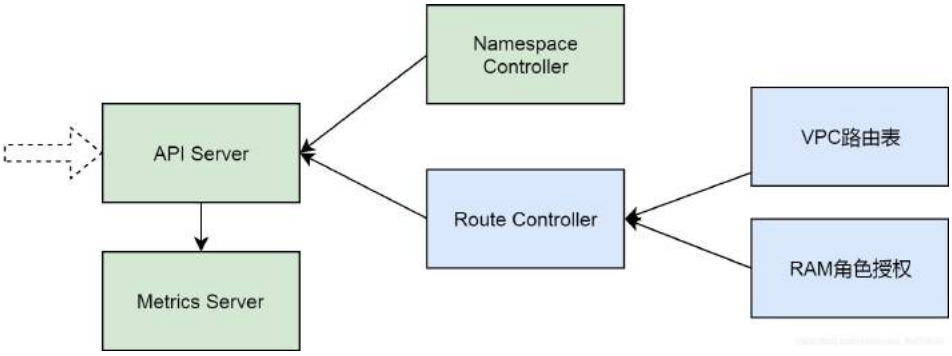
```
58      {
59          "Action": [
60              "vpc:*"
61          ],
62          "Resource": [
63              "*"
64          ],
65          "Effect": "Deny"
66      },
```

当我们把 Effect 修改成 Allow 之后，没多久，所有的 Terminating 状态的 namespace 全部都消失了。

```
[root@i2af6ivvellekm3xr4o4fr2 ~]# kubectl get ns
NAME              STATUS        AGE
catalog           Active        20d
default           Active        23d
dev               Terminating   2d2h
istio-system      Active        11d
kube-public       Active        23d
kube-system       Active        23d
space             Terminating   2d2h
tobedeleteda      Terminating   2d1h
tobedeletedb      Terminating   2d1h
tobedeletedc      Terminating   2d
vk                Active        9d
[root@i2af6ivvellekm3xr4o4fr2 ~]# kubectl get ns
NAME              STATUS   AGE
catalog           Active   20d
default           Active   23d
istio-system      Active   11d
kube-public       Active   23d
kube-system       Active   23d
vk                Active   9d
```


问题大图

总体来说，这个问题与 K8S 集群的 6 个组件有关系，分别是 API Server 及其扩展 Metrics Server，Namespace Controller 和 Route Controller，以及 VPC 路由表和 RAM 角色授权。



通过分析前三个组件的行为，我们定位到，集群网络问题导致了 API Server 无法连接到 Metrics Server；通过排查后三个组件，我们发现导致问题的根本原因是 VPC 路由表被删除且 RAM 角色授权策略被改动。

后记

K8S 集群命名空间删除不掉的问题，是线上比较常见的一个问题。这个问题看起来无关痛痒，但实际上不仅复杂，而且意味着集群重要功能的缺失。这篇文章全面分析了一个这样的问题，其中的排查方法和原理，希望对大家排查类似问题有一定的帮助。

阿里云 ACK 产品安全组配置管理

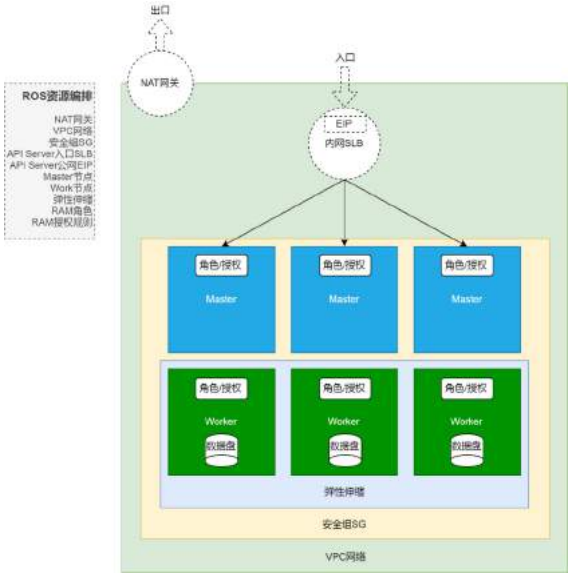
阿里云容器产品 Kubernetes 版本，即 ACK，基于阿里云 IaaS 层云资源创建。资源包括云服务器 ECS，专有网络 VPC，弹性伸缩 ESS 等。以这些资源为基础，ACK 产品实现了 Kubernetes 集群的节点，网络，自动伸缩等组件和功能。

一般而言，用户对 ACK 产品有很大的管理权限，这包括集群扩容，创建服务等。与此同时，用户可以绕过 ACK 产品，对集群底层云资源进行修改。如释放 ECS，删除 SLB。如果不能理清背后的影响，这样的修改会损坏集群功能。

这篇文章会以 ACK 产品安全组的配置管理为核心，深入讨论安全组在集群中扮演的角色，安全组在网络链路中所处的位置，以及非法修改安全组会产生的各类问题。文章内容适用于专有集群和托管集群。

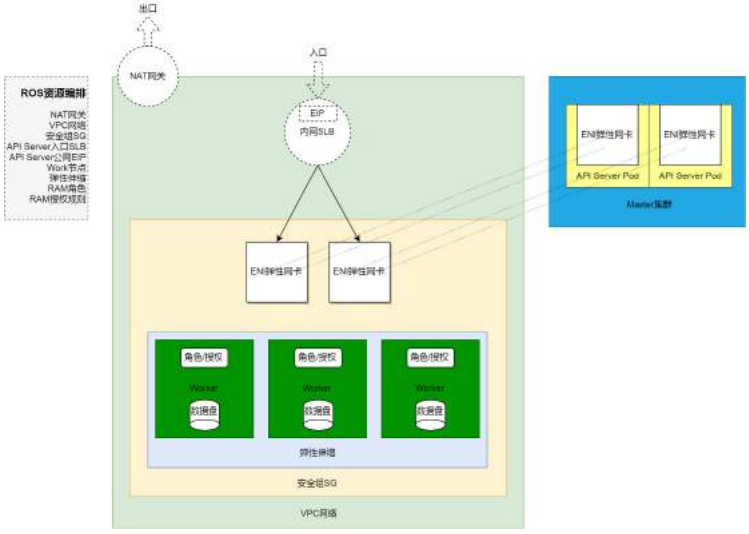
安全组在 ACK 产品中扮演的角色

阿里云 ACK 产品有两种基本形态，专有集群和托管集群。这两种形态的最大差别，就是用户对 master 的管理权限。对安全组来说，两种形态的集群略有差别，这里分开讨论。



专有集群使用资源编排 ROS 模板搭建集群的主框架。其中专有网络是整个集群运行的局域网，云服务器构成集群的节点，安全组构成集群节点的出入防火墙。

另外，集群使用弹性伸缩实现动态扩缩容功能，NAT 网关作为集群的网络出口，SLB 和 EIP 实现集群 API Server 的入口。



托管集群与专有集群类似，同样使用资源编排模板搭建集群的主框架。在托管集群中，云服务器，专有网络，负载均衡 SLB，EIP，安全组等扮演的角色专有集群类似。

与专有集群不同的是，托管集群的 master 系统组件以 Pod 的形式运行在管控集群里。这就是用 Kubernetes 管理 Kubernetes 的概念。

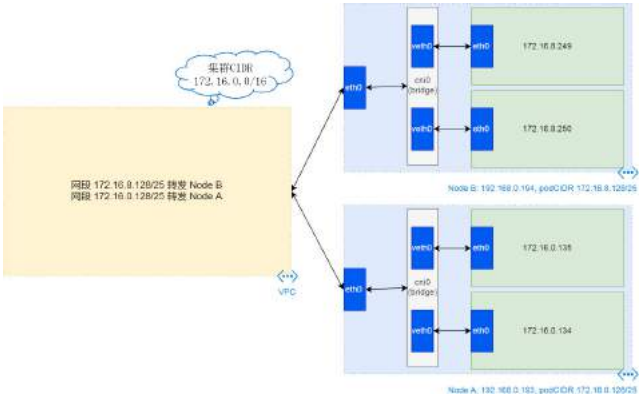
因为托管集群在用户的 VPC 里，而管控集群在阿里云生产账号的 VPC 里。所以这样的架构需要解决的一个核心问题，就是跨账号跨 VPC 通信问题。

为了解决这个问题，此处用到类似传送门的技术。托管集群会在集群 VPC 里创建两个弹性网卡，这两个弹性网卡可以像普通云服务器一样，和集群节点通信。但是这两个网卡被挂载到托管集群的 API Server Pod 上，这就解决了跨 VPC 通信问题。

安全组与 ACK 集群网络

上一节总结了两种形态 ACK 集群的组成原理，以及安全组在集群中所处的位置。简单来说，安全组就是管理网络出入流量的防火墙。

安全组规则是基于数据包的目的地址而限流的。出规则需要基于对目标地址的管控需求而定，而入规则则需要对集群内部通信对象有所理解。以下图为例，ACK 集群的内部的通信对象包括集群节点，和部署在集群上的容器组 Pod 两种。

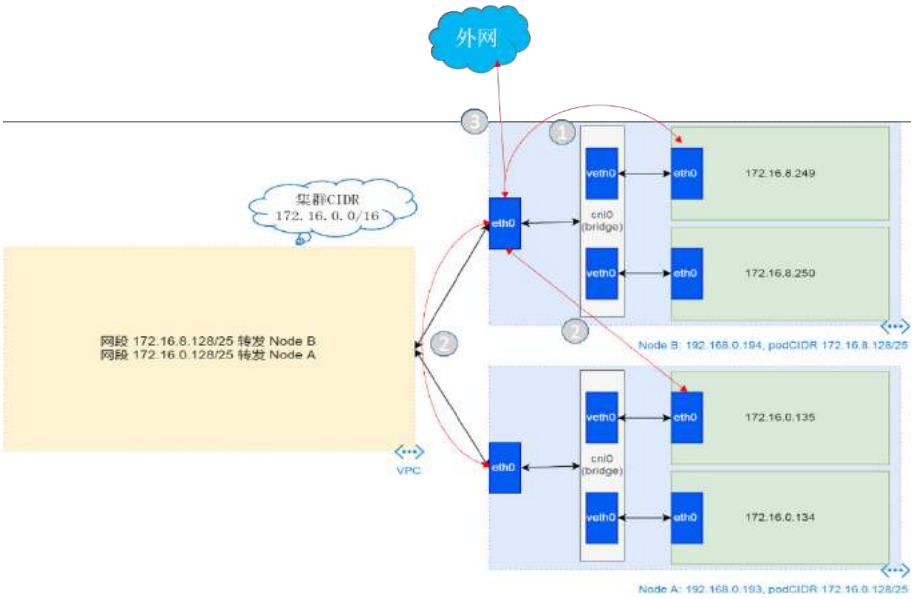


云服务器没有太多特殊的地方，仅仅是简单的连接在 VPC 局域网内的 ECS。而容器组 Pod 是连接在，基于 veth 网口对、虚拟网桥、以及 VPC 路由表所搭建的、和 VPC 独立的虚拟三层网络上的。

总结一下，有两种通信实体和三种通信方式，共六种通信场景。

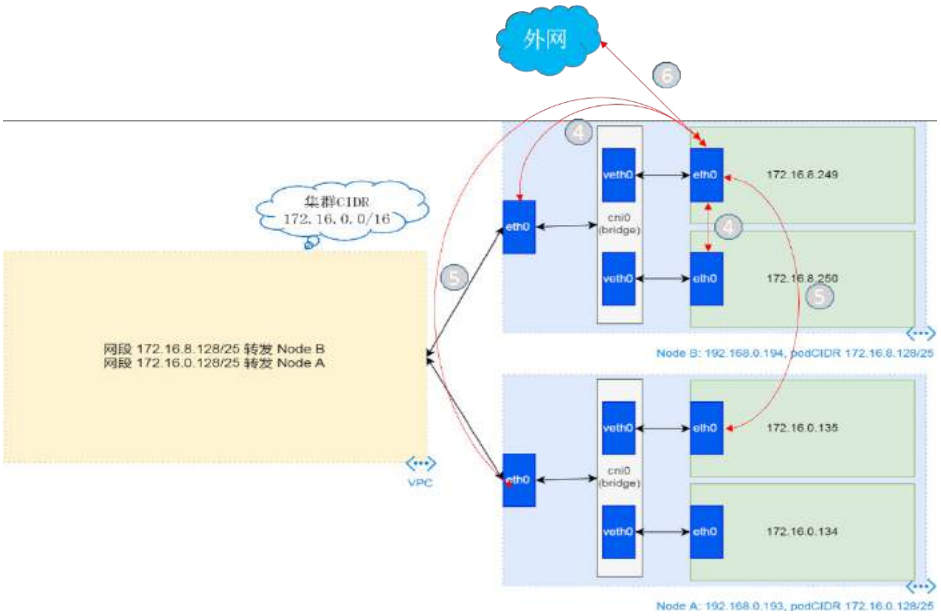
	相同节点通信	跨节点通信	外部通信
节点	无关	无关	有关
Pod	无关	无关 / 有关	有关

前三种场景，以节点为通信实体。第一种场景是节点与其上 Pod 通信，这种场景和安全组无关；第二种场景是节点与其他节点以及 Pod 通信，这种场景下，因为节点在相同 VPC 下，且 Pod 访问 Pod 网段以外的地址都会经过 SNAT，所以和安全组无关；第三种场景是节点与 VPC 之外实体通信，这种情况不管出入都与安全组有关。



后三种场景，以容器组 Pod 为主要通信实体。第四种场景是 Pod 在节点内部与 Pod 和 ECS 通信，这种场景和安全组无关；第五种场景是 Pod 跨节点与其他节点以

及 Pod 通信，这种场景下，如果源地址和目的地址都是 Pod，则需要安全组入规则放行，其他情况与场景二类似；第六种场景是 Pod 与 VPC 之外实体通信，这与场景三类似。



虽然以上场景有些复杂，但是经过总结会发现，与安全组有关的通信，从根本上说就两种情况。一种是 Pod 之间跨节点通信，另一种是节点或 Pod 与外网互访。这里的外网可以是公网，也可以是与集群互联互通的 IDC 或者其他 VPC。

怎么样管理 ACK 集群的安全组规则

上一节详细分析了安全组在 ACK 集群通信的时候，会影响到的场景。最后的结论是，配置 ACK 集群的安全组，只须考虑两种情况，一个是 Pod 跨节点互访，一个是集群和外网互访。

ACK 集群在创建的时候，默认添加了 Pod 网段放行入规则，与此同时保持出规则对所有地址全开。这使得 Pod 之间互访没有问题，同时 Pod 或节点可以随意访问

集群以外的网络。

而在默认规则的基础上对集群安全组的配置管理，其实就是在不影响集群功能的情况下，收紧 Pod 或节点访问外网的能力，和放松集群以外网络对集群的访问。

下边我们分三个常见的场景，来进一步分析，怎么样在默认规则的基础上，进一步管理集群的安全组规则。第一个场景是限制集群访问外网，第二个场景是 IDC 与集群互访，第三个场景是使用新的安全组管理部分节点。

限制集群访问外网

这是非常常见的一个场景。为了在限制集群访问外网的同时，不影响集群本身的功能，配置需要满足三个条件。

1. 不能限制出方向 Pod 网段
2. 不能限制集群访问阿里云云服务的内网地址段 100.64.0.0/10
3. 不能限制集群访问一部分阿里云云服务的公网地址

ecs.cn-hangzhou.aliyuncs.com

ecs-cn-hangzhou.aliyuncs.com

vpc.cn-hangzhou.aliyuncs.com

slb.cn-hangzhou.aliyuncs.com

location-readonly.aliyuncs.com

location.aliyuncs.com

pvtz.cn-hangzhou.aliyuncs.com

cs.cn-hangzhou.aliyuncs.com

nas.cn-hangzhou.aliyuncs.com

oss-cn-hangzhou.aliyuncs.com

cr.cn-hangzhou.aliyuncs.com

metrics.cn-hangzhou.aliyuncs.com

ess.cn-hangzhou.aliyuncs.com

eci.cn-hangzhou.aliyuncs.com

alidns.cn-hangzhou.aliyuncs.com

sls.cn-hangzhou.aliyuncs.com

arms.cn-hangzhou.aliyuncs.com

其中第一条显而易见，第二条为了确保集群可以通过内网访问 DNS 或者 OSS 这类服务，第三条是因为集群在实现部分功能的时候，会通过公网地址访问云服务。

IDC 与集群互访

IDC 与集群互访这种场景，假设 IDC 和集群 VPC 之间，已经通过底层的网络产品打通，IDC 内部机器和集群节点或者 Pod 之间，可以通过地址找到对方。

这种情况下，只需要在确保出方向规则放行 IDC 机器网段的情况下，对入规则配置放行 IDC 机器地址段即可。

使用新的安全组管理节点

某些时候，用户需要新增加一些安全组来管理集群节点。比较典型的用法，包括把集群节点同时加入到多个安全组里，和把集群节点分配给多个安全组管理。

如果把节点加入到多个安全组里，那么这些安全组会依据优先级，从高到低依次匹配规则，

这会给配置管理增加复杂度。而把节点分配给多个安全组管理，则会出现脑裂问题，需要通过安全组之间授权，或者增加规则的方式，确保集群节点之间互通。

典型问题与解决方案

前边的内容包括了安全组在 ACK 集群中所扮演的角色，安全组与集群网络，以及安全组配置管理方法。最后一节基于阿里云售后线上客户海量问题的排查经验，分享一些典型的，与安全组错误配置有关系的问题和解决方案。

使用多个安全组管理集群节点

托管集群默认把节点 ECS 和管控 ENI 弹性网卡放在同一个安全组里，根据安全组的特性，这保证了 ENI 网卡和 ECS 的网卡之间在 VPC 网络平面上的互通。如果把节点从集群默认安全组里移除并纳入其他安全组的管理当中，这导致集群管控 ENI 和节点 ECS 之间无法通行。

这个问题的现象，比较常见的有，使用 kubectl exec 命令无法进去 pod 终端做管理，使用 kubectl logs 命令无法查看 pod 日志等。其中 kubectl exec 命令所返回的报错比较清楚，即从 API Server 连接对应节点 10250 端口超时，这个端口的监听者就是 kubelet。

```
[root@izuf6380y7iwd3qtr1pg21z ~]# kubectl exec -ti -n kube-system  nginx-ingress-controller-565ffc77f4-2fsfm sh
Error from server: error dialing backend: dial tcp 192.168.0.97:10250: i/o timeout
[root@izuf6380y7iwd3qtr1pg21z ~]# kubectl logs -n kube-system  nginx-ingress-controller-565ffc77f4-2fsfm sh
Error from server (BadRequest): container sh is not valid for pod nginx-ingress-controller-565ffc77f4-2fsfm
[root@izuf6380y7iwd3qtr1pg21z ~]# kubectl top nodes
Error from server (ServiceUnavailable): the server is currently unable to handle the request (get nodes.metrics.k8s.io)
```

此问题的解决方案有三种，一个是将集群节点重新加入集群创建的安全组，另一个是对节点所在的安全组和集群创建的安全组之间互相授权，最后一个方式是，在两个安全组里使用规则来互相放行节点 ECS 和管控 ENI 的地址段。

限制集群访问公网或者运营级 NAT 保留地址

专有或托管集群的系统组件，如 cloud controller manager，metrics server，cluster auto scaler 等，使用公网地址或运营商级 NAT 保留地址（100.64.0.0/10）访问阿里云云产品，这些产品包括但不限于负载均衡 SLB，弹性伸缩 ESS，对象存储 OSS。如果安全组限制了集群访问这些地址，则会导致系统组件功能受损。

这个问题的现象，比较常见的有，创建服务的时候，cloud controller manager 无法访问集群节点 metadata 并获取 token 值。集群节点以及其上的系统组件通过节点绑定的授权角色访问云资源，如访问不到 token，会导致权限问题。

```
E0216 07:14:30.977928       1 clientmgr.go:116] token retrieve: role name: Get http://100.100.100.200/latest/meta-data/ram/security-credentials/: dial tcp 100.100.100.200:80: i/o timeout
```

另外一个现象是，集群无法从阿里云镜像仓库下载容器镜像，导致 pod 无法创建。在报错中有明显的，访问阿里云镜像仓库的报错。

```
Events:
  Type     Reason     Age                  From                              Message
  ----     ------     ----                 ----                              -------
  Normal   Scheduled  41m                  default-scheduler                 Successfully assigned kube-system/metrics-server-644f9c977b-d2bck to cn-shanghai.
192.168.0.92
  Warning  Failed     35m (x4 over 40m)    kubelet, cn-shanghai.192.168.0.92  Failed to pull image "registry-vpc.cn-shanghai.aliyuncs.com/acs/metrics-server:v0
.2.1-9dd9511-aliyun": rpc error: code = Unknown desc = Error response from daemon: Get https://registry-vpc.cn-shanghai.aliyuncs.com/v2/: dial tcp: lookup regis
try-vpc.cn-shanghai.aliyuncs.com: no such host
  Warning  Failed     35m (x4 over 40m)    kubelet, cn-shanghai.192.168.0.92  Error: ErrImagePull
  Normal   BackOff    16m (x78 over 40m)   kubelet, cn-shanghai.192.168.0.92  Back-off pulling image "registry-vpc.cn-shanghai.aliyuncs.com/acs/metrics-server:
v0.2.1-9dd9511-aliyun"
  Warning  Failed     6m3s (x116 over 40m)  kubelet, cn-shanghai.192.168.0.92  Error: ImagePullBackOff
```

此问题的解决方案，是在限制集群出方向的时候，确保运营商级 NAT 保留地址 100.64.0.0/10 网段以及阿里云云服务公网地址被放行。其中运营商保留地址比较容易处理，云服务公网地址比较难处理，原因有两个，一个是集群会访问多个云服务且这些云服务的公网地址有可能会更改，另一个是这些云服务可能使用 DNS 负载均衡。所以需要多次解析这些服务的 url 并找出所有 ip 地址并放行。

容器组跨节点通信异常

集群创建的时候，会在安全组里添加容器组网段入方向放行规则。有了这个规则，即使容器组网段和 VPC 网段不一样，容器组在跨节点通信的时候，也不会受到安全组的限制。如果这个默认规则被移除，那么容器组跨节点通信会失败，进而使得多种集群基础功能受损。

这个问题的现象，比较常见的有，容器组 DNS 解析失败，容器组访问集群内部其他服务异常等。如下图，在容器组网段规则被移除之后，从 disk controller 里访问 www.aliyun.com 则无法解析域名，telnet coredns 的地址不通。地址之所以可以 ping 通的原因，是安全组默认放行了所有 icmp 数据。

```
[root@iZuf6b6l253njp6xwsb7voZ ~]# kubectl get pods -n kube-system -o wide | grep dns
coredns-8494f7d9f6-gw2xd                            1/1     Running            0          4d4h    172.20.1.3     cn-shanghai.192.168.0.90   <none>           <non
e>
coredns-8494f7d9f6-stkmm                            1/1     Running            0          4d4h    172.20.1.2     cn-shanghai.192.168.0.90   <none>           <non
e>
[root@iZuf6b6l253njp6xwsb7voZ ~]# kubectl exec -ti -n kube-system alicloud-disk-controller-7476ff6bc4-qqqvt sh
/ # ping www.aliyun.com
ping: bad address 'www.aliyun.com'
/ # ping 172.20.1.3
PING 172.20.1.3 (172.20.1.3): 56 data bytes
64 bytes from 172.20.1.3: seq=0 ttl=62 time=0.260 ms
64 bytes from 172.20.1.3: seq=1 ttl=62 time=0.239 ms
^C
--- 172.20.1.3 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.239/0.249/0.260 ms
/ # telnet 172.20.1.3 53
^C
/ #
```

此问题解决方案比较简单，就是重新把容器组地址段加入安全组。这类问题的难点在于，其引起的问题非常多，现象千奇百怪，所以从问题现象定位到容器组跨节点通信，是解决问题的关键一步。

结束语

这篇文章从三个方面，深入讨论了阿里云 ACK 产品安全组配置管理。这三个方面分别安全组在集群中扮演的角色，安全组与集群网络，以及常见问题和解决方案。

同时通过分析，可以看到 ACK 产品安全组配置管理的三个重点，分别是集群的外网访问控制，集群容器组之间跨节点访问，以及集群使用多个安全组管理。与这三个重点对于的，就是三类常见的问题。

以上总结会在集群创建之前和创建之后，对集群安全组的规划管理有一定指导意义。

二分之一活的微服务

简介：Istio is the future！基本上，我相信对云原生技术趋势有些微判断的同学，都会有这个觉悟。其背后的逻辑其实是比较简单的：当容器集群，特别是 K8S 成为事实上的标准之后，应用必然会不断的复杂化，服务治理肯定会成为强需求。

Istio is the future！基本上，我相信对云原生技术趋势有些微判断的同学，都会有这个觉悟。其背后的逻辑其实是比较简单的：当容器集群，特别是 K8S 成为事实上的标准之后，应用必然会不断的复杂化，服务治理肯定会成为强需求。

Istio 的现状是，聊的人很多，用的人其实很少。所以导致我们能看到的文章，讲道理的很多，讲实际踩坑经验的极少。

阿里云售后团队作为一线踩坑团队，分享问题排查经验，我们责无旁贷。这篇文章，我就跟大家聊一个简单 Istio 问题的排查过程，权当抛砖。

二分之一活的微服务

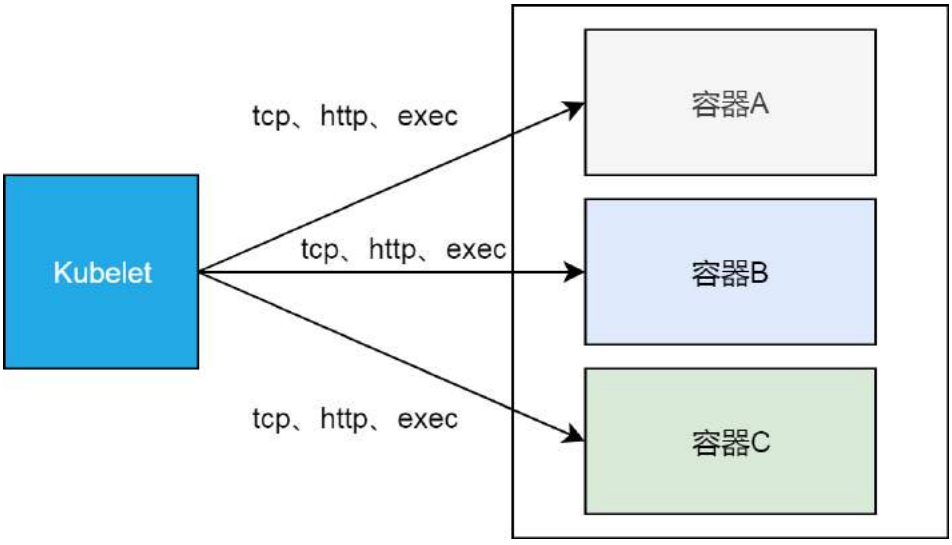
问题是这样的，用户在自己的测试集群里安装了 Istio，并依照官方文档部署 bookinfo 应用来上手 Istio。部署之后，用户执行 kubectl get pods 命令，发现所有的 pods 都只有二分之一个容器是 READY 的。

```
# kubectl get pods
NAME READY STATUS RESTARTS AGE
details-v1-68868454f5-94hzd 1/2 Running 0 1m
productpage-v1-5cb458d74f-28nlz 1/2 Running 0 1m
ratings-v1-76f4c9765f-gjjsc 1/2 Running 0 1m
reviews-v1-56f6855586-dplsf 1/2 Running 0 1m
reviews-v2-65c9df47f8-zdgbw 1/2 Running 0 1m
reviews-v3-6cf47594fd-cvrtf 1/2 Running 0 1m
```

如果从来都没有注意过 READY 这一列的话，我们大概会有两个疑惑：2 在这里是什么意思，以及 1/2 到底意味着什么。

简单来讲，这里的 READY 列，给出的是每个 pod 内部容器的 readiness，即就绪状态。每个集群节点上的 kubelet 会根据容器本身 readiness 规则的定义，分别是 tcp、http 或 exec 的方式，来确认对应容器的 readiness 情况。

更具体一点，kubelet 作为运行在每个节点上的进程，以 tcp/http 的方式（节点网络命名空间到 pod 网络命名空间）访问容器定义的接口，或者在容器的 name-space 里执行 exec 定义的命令，来确定容器是否就绪。



这里的 2 说明这些 pod 里都有两个容器，1/2 则表示，每个 pod 里只有一个容器是就绪的，即通过 readiness 测试的。关于 2 这一点，我们下一节会深入讲，这里我们先看一下，为什么所有的 pod 里，都有一个容器没有就绪。

使用 kubectl 工具拉取第一个 details pod 的编排模板，可以看到这个 pod 里两个容器，只有一个定义了 readiness probe。对于未定义 readiness probe 的容器，kubelet 认为，只要容器里的进程开始运行，容器就进入就绪状态了。所以 1/2 个就

绪 pod，意味着，有定义 readiness probe 的容器，没有通过 kubelet 的测试。

没有通过 readiness probe 测试的是 istio-proxy 这个容器。它的 readiness probe 规则定义如下。

```
readinessProbe:
  failureThreshold: 30
  httpGet:
    path: /healthz/ready
    port: 15020
    scheme: HTTP
  initialDelaySeconds: 1
  periodSeconds: 2
  successThreshold: 1
  timeoutSeconds: 1
```

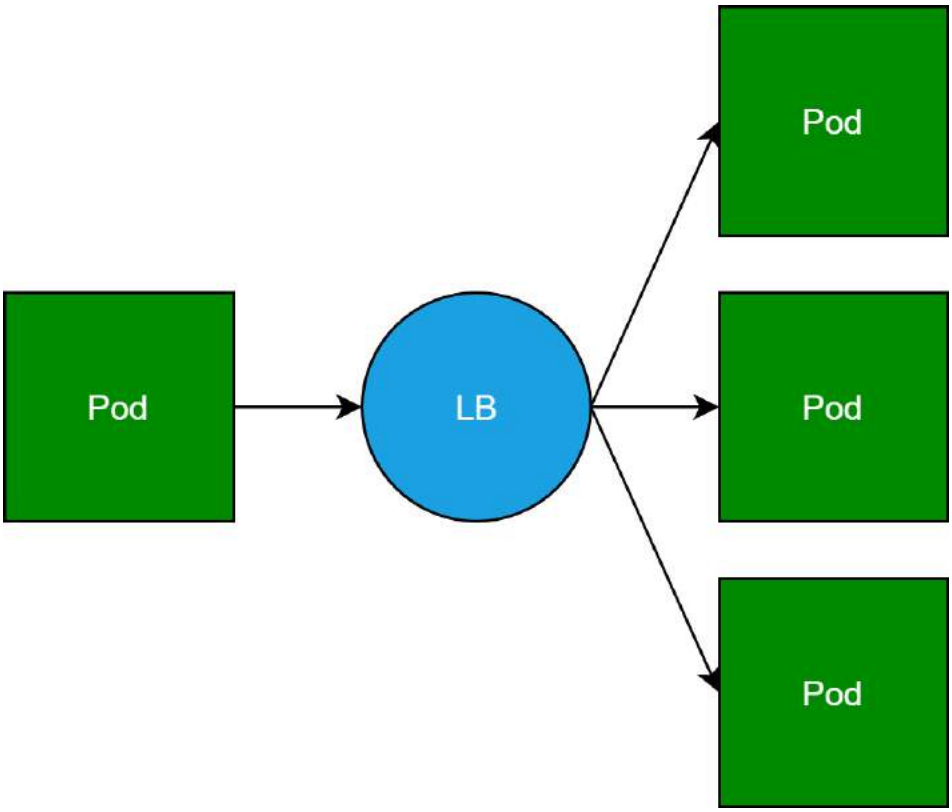
我们登录这个 pod 所在的节点，用 curl 工具来模拟 kubelet 访问下边的 uri，测试 istio-proxy 的就绪状态。

```
# curl http://172.16.3.43:15020/healthz/ready -v
* About to connect() to 172.16.3.43 port 15020 (#0)
*   Trying 172.16.3.43...
* Connected to 172.16.3.43 (172.16.3.43) port 15020 (#0)
> GET /healthz/ready HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 172.16.3.43:15020
> Accept: */*>
< HTTP/1.1 503 Service Unavailable< Date: Fri, 30 Aug 2019 16:43:50 GMT
< Content-Length: 0
< *
Connection #0 to host 172.16.3.43 left intact
```

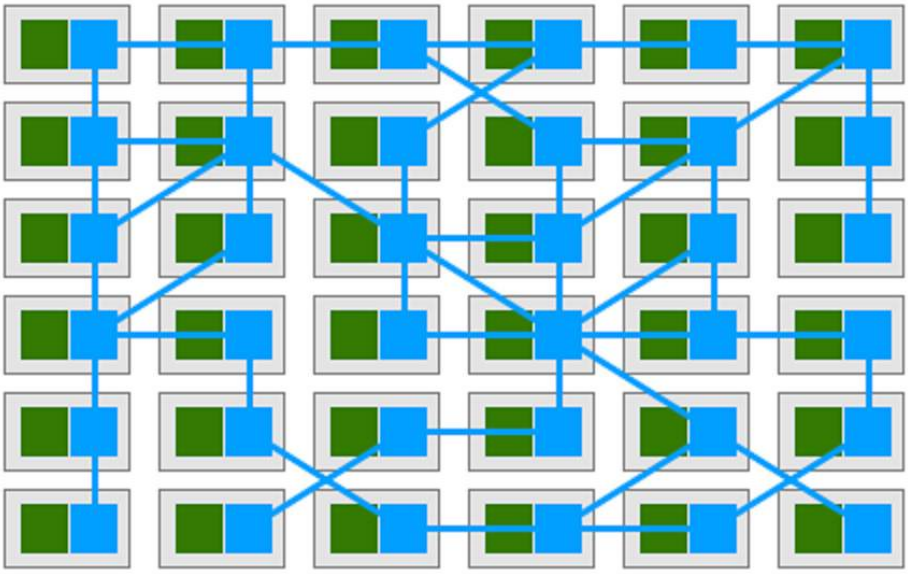
绕不过去的大图

上一节我们描述了问题现象，但是留下一个问题，就是 pod 里的容器个数为什么是 2。虽然每个 pod 本质上至少有两个容器，一个是占位符容器 pause，另一个是真正的工作容器，但是我们在使用 kubectl 命令获取 pod 列表的时候，READY 列是不包括 pause 容器的。

这里的另外一个容器，其实就是服务网格的核心概念 sidercar。其实把这个容器叫做 sidecar，某种意义上是不能反映这个容器的本质的。Sidecar 容器本质上是反向代理，它本来是一个 pod 访问其他服务后端 pod 的负载均衡。



然而，当我们为集群中的每一个 pod，都“随身”携带一个反向代理的时候，pod 和反向代理就变成了服务网格。正如下边这张经典大图所示。这张图实在有点难画，所以只能借用，绕不过去。



所以 sidecar 模式，其实是“自带通信员”模式。这里比较有趣的是，在我们把 sidecar 和 pod 绑定在一块的时候，sidecar 在出流量转发时扮演着反向代理的角色，而在入流量接收的时候，可以做超过反向代理职责的一些事情。这点我们会在其他文章里讨论。

Istio 在 K8S 基础上实现了服务网格，Isito 使用的 sidecar 容器就是第一节提到的，没有就绪的容器。所以这个问题，其实就是服务网格内部，所有的 sidecar 容器都没有就绪。

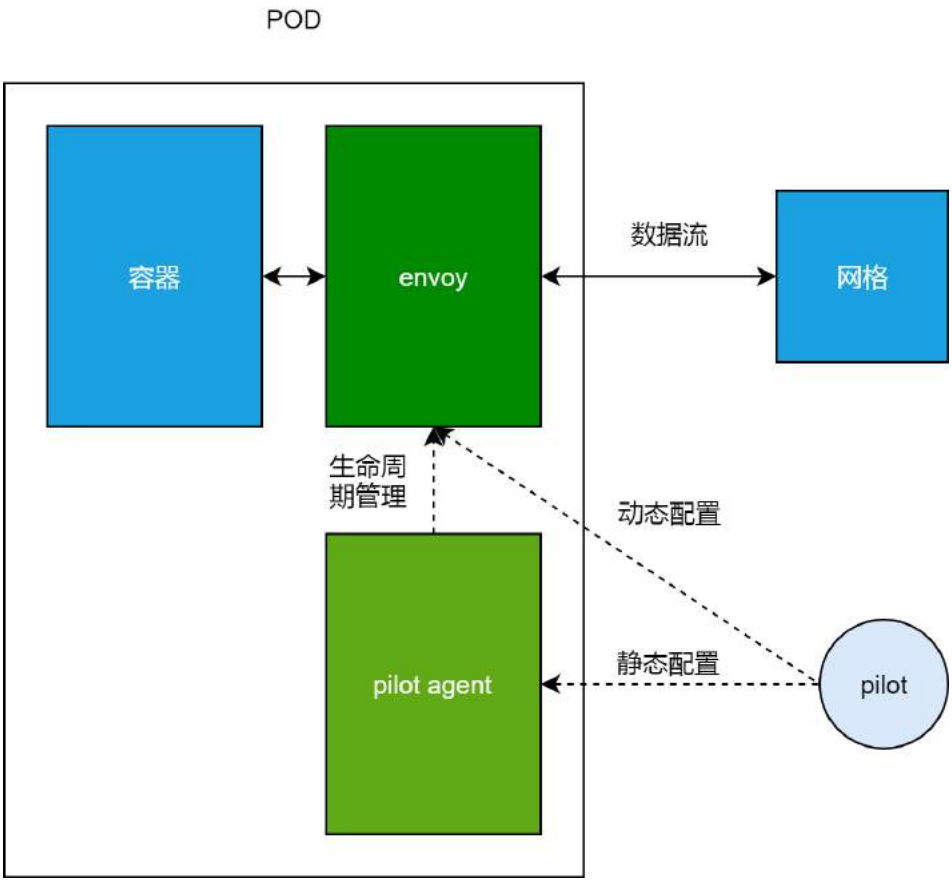
代理与代理的生命周期管理

上一节我们看到，istio 中的每个 pod，都自带了反向代理 sidecar。我们遇到的问题是，所有的 sidecar 都没有就绪。我们也看到 readiness probe 定义的，判断 sidecar 容器就绪的方式就是访问下边这个接口。

```
http://<pod ip>:15020/healthz/ready
```

接下来，我们深入看下 pod，以及其 sidecar 的组成及原理。在服务网格里，一个 pod 内部除了本身处理业务的容器之外，还有 istio-proxy 这个 sidecar 容器。正常情况下，istio-proxy 会启动两个进程，pilot-agent 和 envoy。

如下图，envoy 是实际上负责流量管理等功能的代理，从业务容器出、入的数据流，都必须要经过 envoy；而 pilot-agent 负责维护 envoy 的静态配置，以及管理 envoy 的生命周期。这里的动态配置部分，我们在下一节会展开来讲。



我们可以使用下边的命令进入 pod 的 istio-proxy 容器做进一步排查。这里的一个小技巧，是我们可以以用户 1337，使用特权模式进入 istio-proxy 容器，如此就

可以使用 iptables 等只能在特权模式下运行的命令。

```
docker exec -ti -u 1337 --privileged <istio-proxy container id> bash
```

这里的 1337 用户，其实是 sidecar 镜像里定义的一个同名用户 istio-proxy，默认 sidecar 容器使用这个用户。如果我们在以上命令中，不使用用户选项 u，则特权模式实际上是赋予 root 用户的，所以我们在进入容器之后，需切换到 root 用户执行特权命令。

进入容器之后，我们使用 netstat 命令查看监听，我们会发现，监听 readiness probe 端口 15020 的，其实是 pilot-agent 进程。

```
istio-proxy@details-v1-68868454f5-94hzd:/$ netstat -lnpt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address    Foreign Address   State   PID/Program name
tcp        0      0 0.0.0.0:15090    0.0.0.0:*         LISTEN  19/envoy
tcp        0      0 127.0.0.1:15000  0.0.0.0:*         LISTEN  19/envoy
tcp        0      0 0.0.0.0:9080     0.0.0.0:*         LISTEN  -
tcp6       0      0 :::15020         :::*              LISTEN  1/pilot-agent
```

我们在 istio-proxy 内部访问 readiness probe 接口，一样会得到 503 的错误。

就绪检查的实现

了解了 sidecar 的代理，以及管理代理生命周期的 pilot-agent 进程，我们可以稍微思考一下 pilot-agent 应该怎么去实现 healthz/ready 这个接口。显然，如果这个接口返回 OK 的话，那不仅意味着 pilot-agent 是就绪的，而必须确保代理是工作的。

实际上 pilot-agent 就绪检查接口的实现正是如此。这个接口在收到请求之后，会去调用代理 envoy 的 server_info 接口。调用所使用的的 IP 是 localhost。这个非常好理解，因为这是同一个 pod 内部进程通信。使用的端口是 envoy 的 proxyAdminPort，即 15000。

有了以上的知识准备之后，我们来看下 istio-proxy 这个容器的日志。实际上，在容器日志里，一直在重复输出一个报错，这句报错分为两部分，其中 Envoy proxy is NOT ready 这部分是 pilot agent 在响应 healthz/ready 接口的时候输出的信息，即 Envoy 代理没有就绪；而剩下的 config not received from Pilot (is Pilot running?): cds updates: 0 successful, 0 rejected; lds updates: 0 successful, 0 rejected 这部分，是 pilot-agent 通过 proxyAdminPort 访问 server_info 的时候带回的信息，看起来是 envoy 没有办法从 Pilot 获取配置。

```
Envoy proxy is NOT ready: config not received from Pilot (is Pilot running?):
cds updates: 0 successful, 0
rejected; lds updates: 0 successful, 0 rejected.
```

到这里，建议大家回退看下上一节的插图，在上一节我们选择性的忽略是 Pilot 到 envoy 这条虚线，即动态配置。这里的报错，实际上是 envoy 从控制面 Pilot 获取动态配置失败。

控制面和数据面

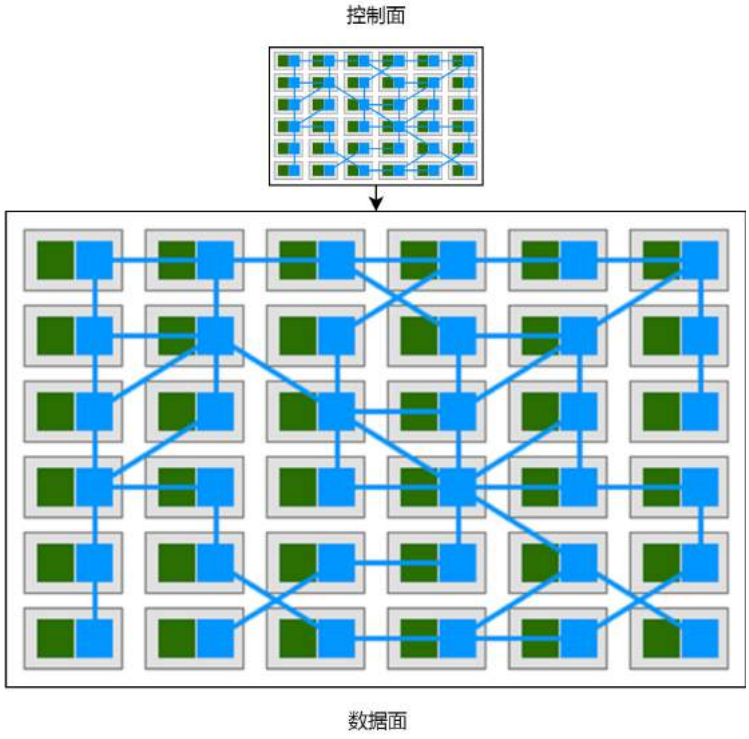
目前为止，这个问题其实已经很清楚了。在进一步分析问题之前，我聊一下我对控制面和数据面的理解。控制面数据面模式，可以说无处不在。我们这里举两个极端的例子。

第一个例子，是 dhcp 服务器。我们都知道，在局域网中的电脑，可以通过配置 dhcp 来获取 ip 地址，这个例子中，dhcp 服务器统一管理，动态分配 ip 地址给网络

中的电脑，这里的 dhcp 服务器就是控制面，而每个动态获取 ip 的电脑就是数据面。

第二个例子，是电影剧本，和电影的演出。剧本可以认为是控制面，而电影的演出，包括演员的每一句对白，电影场景布置等，都可以看做是数据面。

我之所以认为这是两个极端，是因为在第一个例子中，控制面仅仅影响了电脑的一个属性，而第二个例子，控制面几乎是数据面的一个完整的抽象和拷贝，影响数据面的方方面面。Istio 服务网格的控制面是比较靠近第二个例子的情况，如下图。



Istio 的控制面 Pilot 使用 grpc 协议对外暴露接口 istio-pilot.istio-system:15010，而 envoy 无法从 Pilot 处获取动态配置的原因，是在所有的 pod 中，集群 dns 都无法使用。

简单的原因

这个问题的原因其实比较简单，在 sidecar 容器 istio-proxy 里，envoy 不能访问 Pilot 的原因是集群 dns 无法解析 istio-pilot.istio-system 这个服务名字。在容器里看到 resolv.conf 配置的 dns 服务器是 172.19.0.10，这个是集群默认的 kube-dns 服务地址。

```
istio-proxy@details-v1-68868454f5-94hzd:/$ cat /etc/resolv.conf
nameserver 172.19.0.10
search default.svc.cluster.local svc.cluster.local cluster.local localdomain
```

但是客户删除重建了 kube-dns 服务，且没有指定服务 IP，这导致，实际上集群 dns 的地址改变了，这也是为什么所有的 sidecar 都无法访问 Pilot。

```
# kubectl get svc -n kube-system
NAME        TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)         AGE
kube-dns    ClusterIP   172.19.9.54    <none>        53/UDP,53/TCP   5d
```

最后，通过修改 kube-dns 服务，指定 IP 地址，sidecar 恢复正常。

```
# kubectl get pods
NAME READY STATUS RESTARTS AGE
details-v1-68868454f5-94hzd 2/2 Running 0 6d
nginx-647d5bf6c5-gfvkm 2/2 Running 0 2d
nginx-647d5bf6c5-wvfpd 2/2 Running 0 2d
productpage-v1-5cb458d74f-28nlz 2/2 Running 0 6d
ratings-v1-76f4c9765f-gjjsc 2/2 Running 0 6d
reviews-v1-56f6855586-dplsf 2/2 Running 0 6d
reviews-v2-65c9df47f8-zdgbw 2/2 Running 0 6d
reviews-v3-6cf47594fd-cvrtf 2/2 Running 0 6d
```

结论

这其实是一个比较简单的问题，排查过程其实也就几分钟。但是写这篇文章，有点感觉是在看长安十二时辰，短短几分钟的排查过程，写完整背后的原理，前因后果，却花了几个小时。这是 Istio 文章的第一篇，希望在大家排查问题的时候，有所帮助。

半夜两点 Ca 证书过期问题处理惨况总结

简介：11 月 22 号半夜 2 点，被值班同学的电话打醒。了解下来，大概情况是，客户某一台 K8s 集群节点重启之后，他再也无法创建 Istio 虚拟服务和 Pod 了。 一来对 Istio 还不是那么熟悉，二来时间可能有点晚，脑子还在懵圈中，本来一个应该比较轻松解决掉的问题，花了几十分钟看代码，处理的惨不忍睹。

11 月 22 号半夜 2 点，被值班同学的电话打醒。了解下来，大概情况是，客户某一台 K8s 集群节点重启之后，他再也无法创建 Istio 虚拟服务和 Pod 了。

一来对 Istio 还不是那么熟悉，二来时间可能有点晚，脑子还在懵圈中，本来一个应该比较轻松解决掉的问题，花了几十分钟看代码，处理的惨不忍睹。最终还是在某位大神帮助下，解决了问题。

鉴于此问题，以及相关报错，在网上找不到对应的文章，所以这里分享下这个问题，避免后来的同学，在同样的地方踩坑。另外谨以此篇致敬工作中遇到过的大神！

不断重启的 Citadel

Citadel 是 istio 的证书分发中心。证书即某个实体的身份证明，直接代表着实体本身参与信息交流活动。Citadel 作为证书分发中心，负责替服务网格中每个服务创建身份证书，方便服务之间安全交流。

这个问题的现象是，Citadel 再也无法启动了，导致无法创建新的虚拟服务和 Pod 实例。观察 Citadel，发现其不断重启，并输出以下报错信息。

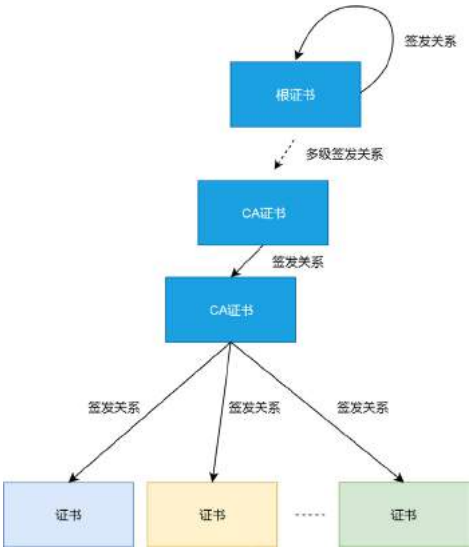
```
2019-11-22T02:40:34.814547Z     warn    Neither --kubeconfig nor --master was
specified.  Using the
inClusterConfig.  This might not work.
2019-11-22T02:40:34.815408Z     info    Use self-signed certificate as the CA
certificate
2019-11-22T02:40:34.840128Z     error   Failed to create a self-signed
Citadel (error: [failed to create CA
KeyCertBundle (cannot verify the cert with the provided root chain and cert
pool)])
```

通过代码分析，以及最后一行报错，可以理解问题背后的逻辑：

1. Citadel 不能启动，是因为其无法创建自签名的证书（Failed to create a self-signed Citadel）
2. Citadel 无法创建自签名证书的原因，又是由于它不能创建秘钥和证书（failed to create CA KeyCertBundle）
3. 而前两条的根本的原因，是因为在验证创建的自签名证书的时候，验证失败（cannot verify the cert with the provided root chain and cert pool）

一般意义上的证书验证

证书的签发关系，会把证书连接成一棵证书签发关系树。顶部是根证书，一般由可信第三方持有，根证书都是自签名的，即其不需要其他机构来对其身份提供证明。其他层级的 CA 证书，都是由上一层 CA 证书签发。最底层的证书，是给具体应用使用的证书。



验证这棵树上的证书，分两种情况：

根证书验证。因为即是签发者，又是被签发者，所以根证书验证，只需要提供根证书本身，一般肯定验证成功。

其他证书验证。需要提供证书本身，以及其上层所有 CA 证书，包括根证书。

自签名证书验证失败 Citadel 启动失败的根本原因，是其创建的自签名的证书，无法验证通过。而这一点，也是我处理问题时，卡壳的原因。左思右想，不明白为什么刚刚新建的自签名证书，都验证不通过。

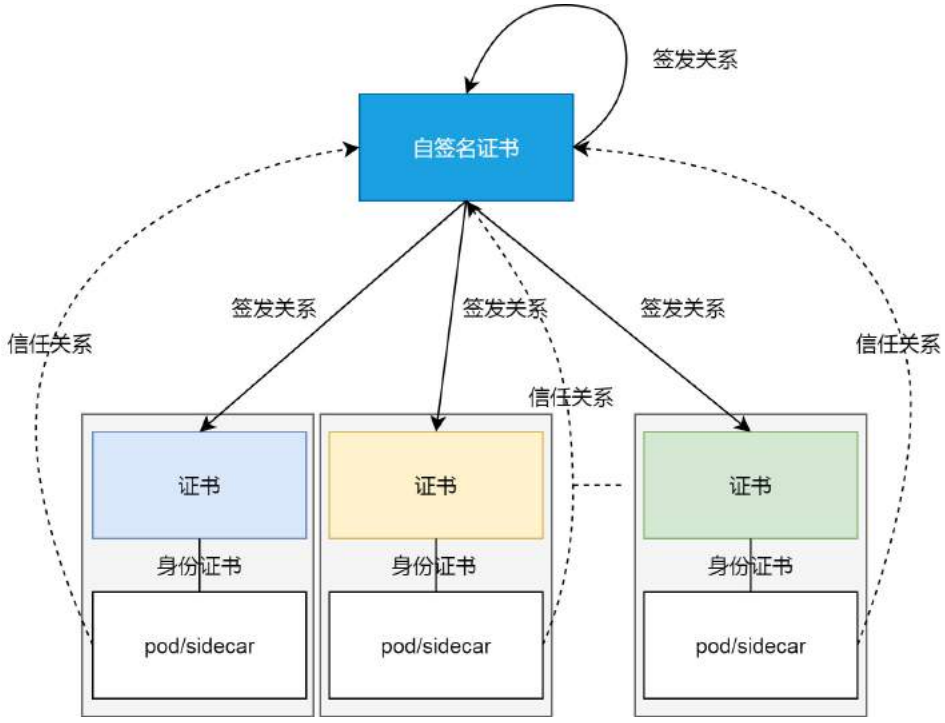
大神定理

有一条定理，就是我们绞尽脑汁，耗费大量时间无法解决的问题，在大神眼里，可能就是几秒钟的事情。11 月 22 号半夜，这条真理再次被验证。

因为实在想不通，但是用户又非常着急，所以最终还是打扰了一位大神，他只大概看了一下报错，就判断是 CA 证书过期问题。使用 istio 提供的证书验证脚本，很快证实了他的判断。相关文章见最后参考部分。

Citadel 证书体系

问题解决了，这里总结下 Citadel 证书体系。大多数用户使用 Isito 的时候，都会选择使用自签名的根证书。自签名根证书，证书，以及证书使用者 sidecar 三种之间有三种关系：



1. 根证书和证书之间的签发关系。这种关系，保证了信任的传递性质。
2. 证书和 sidecar 之间的持有和被持有关系。某种意义上，这是给 pod/sidecar 和证书画上了等号。
3. 根证书和 sidecar 之间的信任关系。这与前两条加起来，sidecar 就信任所有根证书签发的证书。

以上三条，即可保证，在互相通信的时候，pod/sidecar 之间可以完成 tls 双向认证成功。

犯的错

这个问题排查中，实际上犯了两个错误。一个是代码阅读不仔细，一直盯着自签名证书新建的逻辑看。因为有了这个前提，即新建证书验证失败，所以没有办法理解，为什么新建的自签名证书也会验证失败，所以倾向于认为是底层安全库出了问题；另外一个是，只盯着 Citadel 不能启动的报错做，忽略了另外一条线索，就是 Pod 和虚拟服务创建失败。实际上后来发现，pod 和虚拟服务创建失败的报错，有更明显的证书过期错误信息。

> virtualservices.networking.istio.io "xxxx" could not be patched: Internal error occurred: failed calling admission webhook "pilot.validation.istio.io": Post https://istio-galley.istio-system.svc:443/admitpilot?timeout=30s: x509: certificate has expired or is not yet valid.

后记

在 Istio 比较早期的版本中，自签名 Ca 证书有效期只有一年时间，如果使用老版本 Istio 超过一年，就会遇到这个问题。当证书过期之后，我们创建新的虚拟服务或者 pod，都会因为 CA 证书过期而失败。而这时如果 Citadel 重启，它会读取过期证书并验证其有效性，就会出现以上 Cidatel 不能启动的问题。

这个 Ca 证书在 K8s 集群中，是以 istio-ca-secret 命名的 secret，我们可以使用 openssl 解码证书来查看有效期。这个问题比较简单的处理方法，就是删除这个 Secret，并重启 Citadel，这时 Citadel 会走向新建和验证自签名 Ca 证书的逻辑并刷新 Ca 证书。或者参考以下官网处理方式。

参考

https://istio.io/docs/ops/security/root-transition/

Alibaba Group
阿里巴巴集团

阿里云 开发者社区

扫一扫二维码图案，关注我吧

下载更多免费电子书

云服务技术课堂技术圈

云服务技术大学钉钉群

扫码关注阿里巴巴云原生