田飞雨

kube-apiserver 的设计与实现

2020-02-24

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

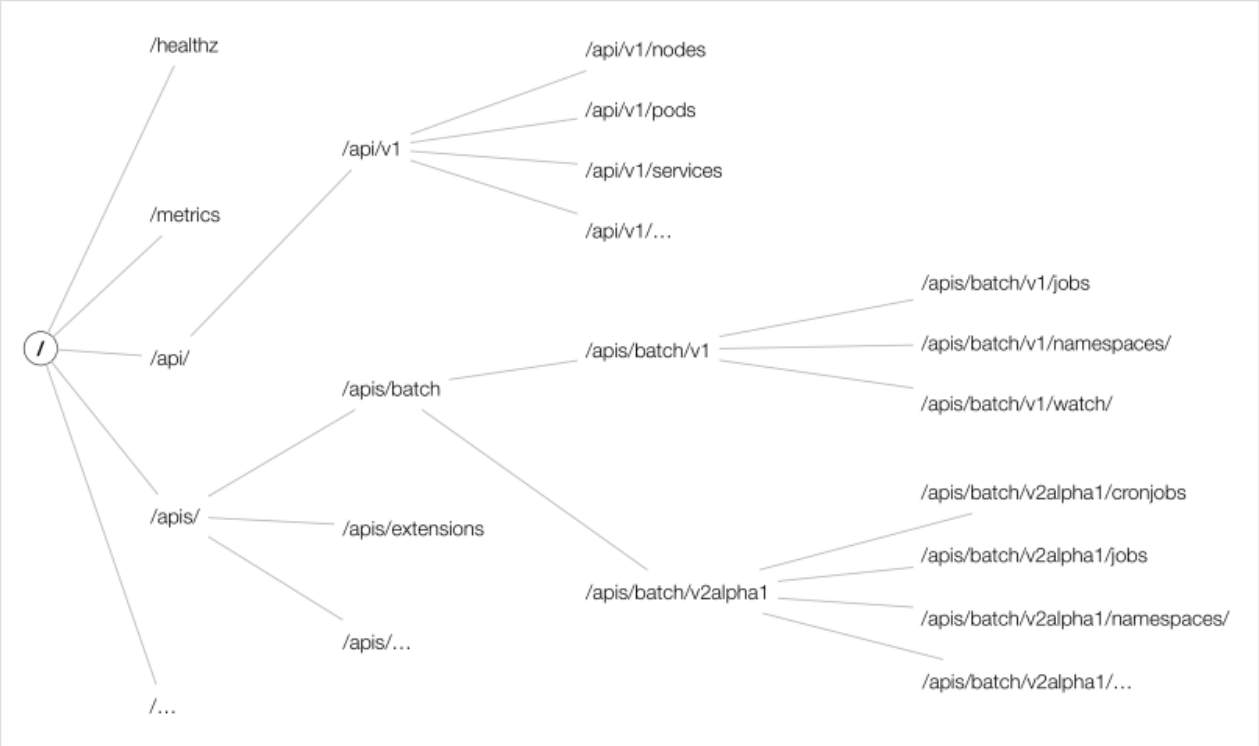
- 提供 Kubernetes API，包括认证授权、数据校验以及集群状态变更等，供客户端及其他组件调用；
- 代理集群中的一些附加组件组件，如 Kubernetes UI、metrics-server、npd 等；
- 创建 kubernetes 服务，即提供 apiserver 的 Service，kubernetes Service；
- 资源在不同版本之间的转换；

kube-apiserver 处理流程

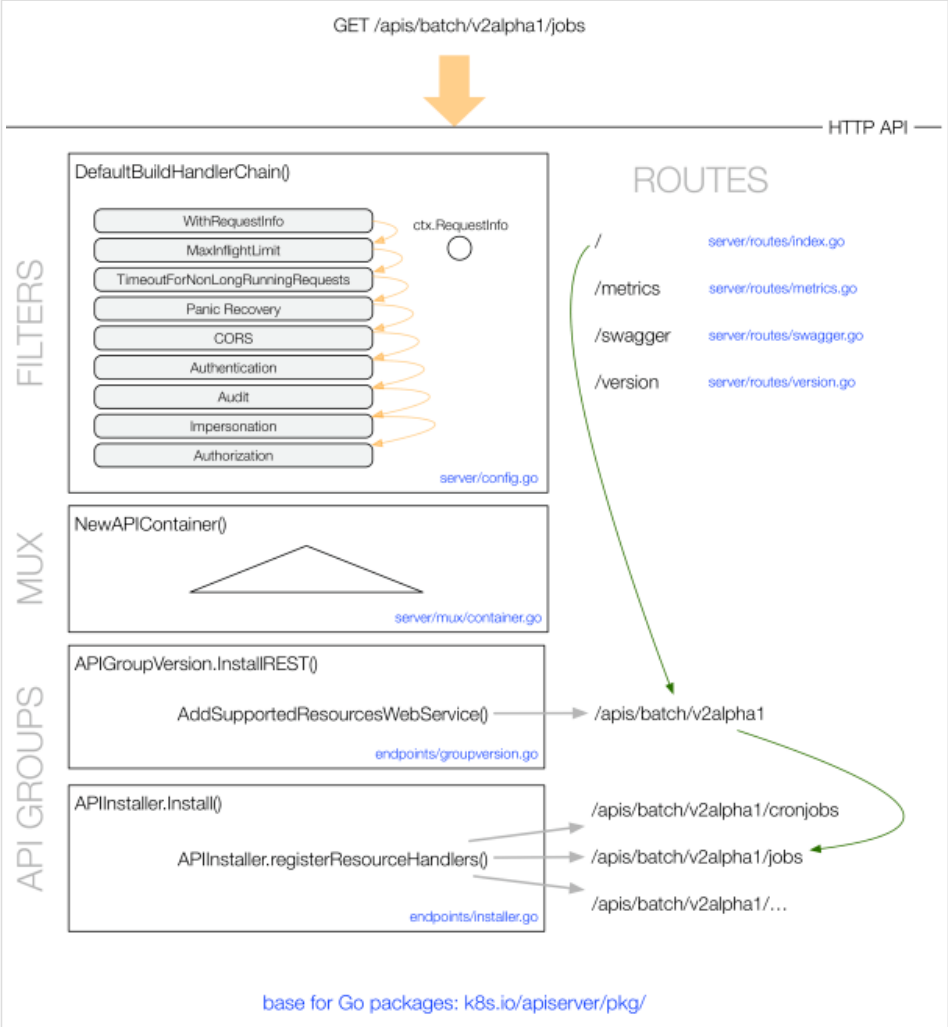
kube-apiserver 主要通过对外提供 API 的方式与其他组件进行交互，可以调用 kube-apiserver 的接口 `$ curl -k https://<masterIP>:6443` 或者通过其提供的 **swagger-ui** 获取到，其主要有以下三种 API：

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

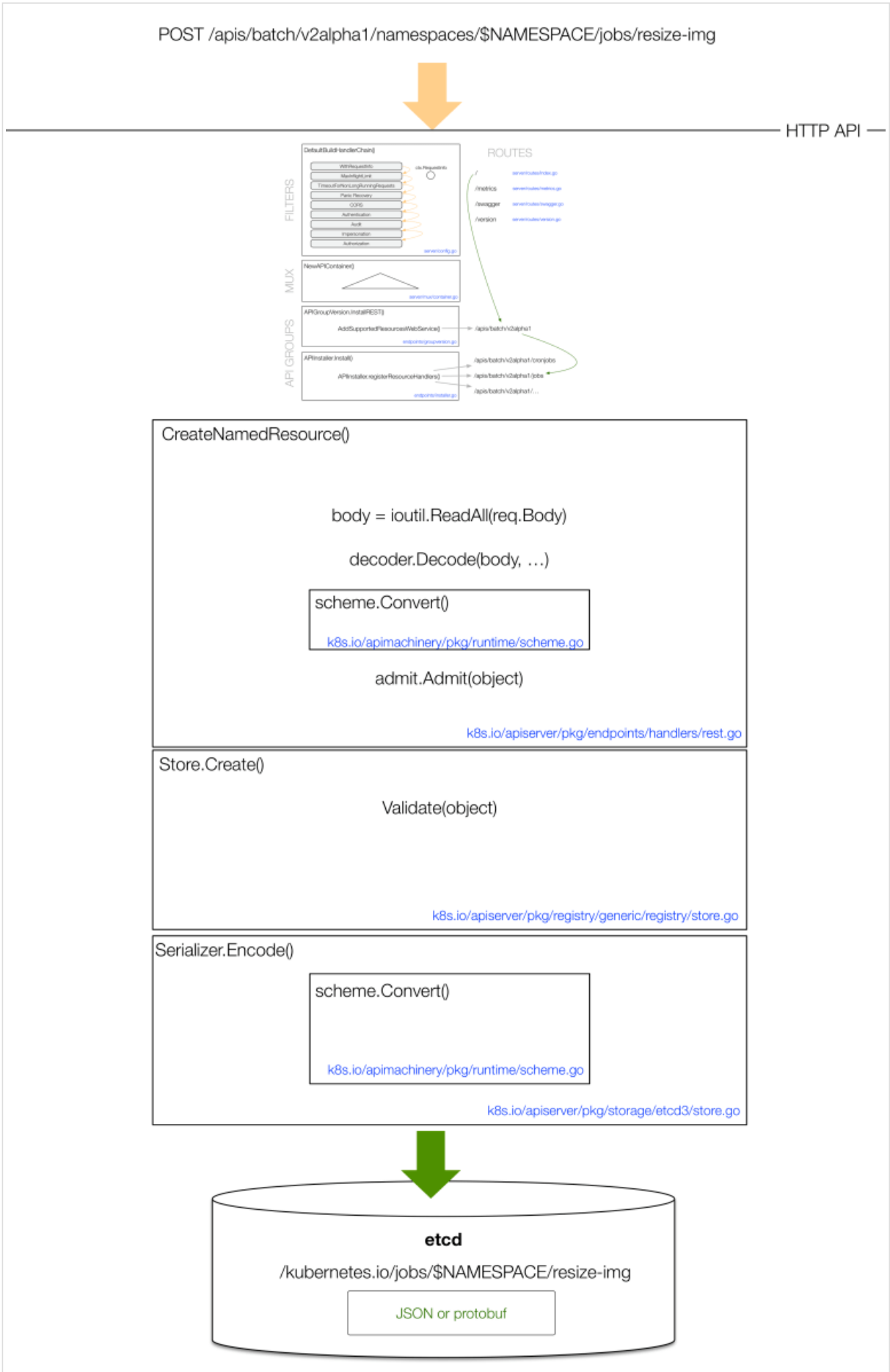
API 的 URL 大致以 `/apis/group/version/namespaces/my-ns/myresource` 组成，其中 API 的结构大致如下图所示：



了解了 kube-apiserver 的 API 后，下面会介绍 kube-apiserver 如何处理一个 API 请求，一个请求完整的流程如下图所示：



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



Decoder

kubernetes 中的多数 resource 都会有一个 internal version ，因为在整个开发过程中一个 resource 可能会对应多个 version，比如 deployment 会有 extensions/v1beta1 ， apps/v1 。 为了避免出现问题，kube-apiserver 必须要知道如何在每一对版本之间进行转换（例如，v1⇔v1alpha1，v1⇔v1beta1，

- 3、 namingController： 检查 crd obj 中是否有命名冲突，可在 crd .status.conditions 中查看；
- 4、 establishingController： 检查 crd 是否处于正常状态，可在 crd .status.conditions 中查看；
- 5、 nonStructuralSchemaController： 检查 crd obj 结构是否正常，可在 crd .status.conditions 中查看；
- 6、 apiApprovalController： 检查 crd 是否遵循 kubernetes API 声明策略，可在 crd .status.conditions 中查看；
- 7、 finalizingController： 类似于 finalizes 的功能，与 CRs 的删除有关；

kube-apiserver 启动流程分析

> kubernetes 版本：v1.16

首先分析 kube-apiserver 的启动方式，kube-apiserver 也是通过其 Run 方法启动主逻辑的，在 Run 方法调用之前会进行解析命令行参数、设置默认值等。

Run

Run 方法的主要逻辑为：

- 1、调用 CreateServerChain 构建服务调用链并判断是否启动非安全的 http server，http server 链中包含 apiserver 要启动的三个 server，以及为每个 server 注册对应资源的路由；
- 2、调用 server.PrepareRun 进行服务运行前的准备，该方法主要完成了健康检查、存活检查和 OpenAPI 路由的注册工作；
- 3、调用 prepared.Run 启动 https server；

server 的初始化使用委托模式，通过 DelegationTarget 接口，把基本的 API Server、CustomResource、Aggregator 这三种服务采用链式结构串联起来，对外提供服务。

k8s.io/kubernetes/cmd/kube-apiserver/app/server.go:147

```
1  func Run(completeOptions completedServerRunOptions, stopCh <-chan struct{}) error {
2      server, err := CreateServerChain(completeOptions, stopCh)
3      if err != nil {
4          return err
5      }
6
7      prepared, err := server.PrepareRun()
8      if err != nil {
9          return err
10     }
11
12     return prepared.Run(stopCh)
13 }
```

CreateServerChain

CreateServerChain 是完成 server 初始化的方法，里面包含 APIExtensionsServer 、 KubeAPIServer 、 AggregatorServer 初始化的所有流程，最终返回 aggregatorapiserver.APIAggregator 实例，初始化流程主要有：http filter chain 的配置、API Group 的注册、http path 与 handler 的关联以及 handler 后端存储 etcd 的配置。其主要逻辑为：

- 1、调用 CreateKubeAPIServerConfig 创建 KubeAPIServer 所需要的配置，主要是创建 master.Config ，其中会调用 buildGenericConfig 生成 genericConfig，genericConfig 中包含 apiserver 的核心配置；
- 2、判断是否启用了扩展的 API server 并调用 createAPIExtensionsConfig 为其创建配置，apiExtensions server 是一个代理服务，用于代理 kubeapiserver 中的其他 server，比如 metric-server；

```
45          insecureHandlerChain := kubeserver.BuildInsecureHandlerChain(aggregatorServer.GenericAPIServer.UnprotectedHandler(), kubeAPIS
46          if err := insecureServingInfo.Serve(insecureHandlerChain, kubeAPIServerConfig.GenericConfig.RequestTimeout, stopCh); err != nil {
47              return nil, err
48          }
49      }
50      return aggregatorServer, nil
51  }
```

CreateKubeAPIServerConfig

在 CreateKubeAPIServerConfig 中主要是调用 buildGenericConfig 创建 genericConfig 以及构建 master.Config 对象。

k8s.io/kubernetes/cmd/kube-apiserver/app/server.go:271

```
1   func CreateKubeAPIServerConfig(
2       s completedServerRunOptions,
3       nodeTunneler tunneler.Tunneler,
4       proxyTransport *http.Transport,
5   ) (.....) {
6
7       // 1、构建 genericConfig
8       genericConfig, versionedInformers, insecureServingInfo, serviceResolver, pluginInitializers, admissionPostStartHook, storageFactory,  la
9       if lastErr != nil {
10          return
11      }
12
13      ......
14
15      // 2、初始化所支持的 capabilities
16      capabilities.Initialize(capabilities.Capabilities{
17          AllowPrivileged: s.AllowPrivileged,
18          PrivilegedSources: capabilities.PrivilegedSources{
19              HostNetworkSources: []string{},
20              HostPIDSources:    []string{},
21              HostIPCSources:    []string{},
22          },
23          PerConnectionBandwidthLimitBytesPerSec: s.MaxConnectionBytesPerSec,
24      })
25
26      // 3、获取 service ip range 以及 api server service IP
27      serviceIPRange, apiServerServiceIP, lastErr := master.DefaultServiceIPRange(s.PrimaryServiceClusterIPRange)
28      if lastErr != nil {
29          return
30      }
31
32      ......
33
34      // 4、构建 master.Config 对象
35      config = &master.Config{......}
36
37      if nodeTunneler != nil {
38          config.ExtraConfig.KubeletClientConfig.Dial = nodeTunneler.Dial
39      }
40      if config.GenericConfig.EgressSelector != nil {
41          config.ExtraConfig.KubeletClientConfig.Lookup = config.GenericConfig.EgressSelector.Lookup
42      }
43
44      return
45  }
```

buildGenericConfig

主要逻辑为：

- 1、调用 genericapiserver.NewConfig 生成默认的 genericConfig，genericConfig 中主要配置了 DefaultBuildHandlerChain ，DefaultBuildHandlerChain 中包含了认证、鉴权等一系列 http filter chain；
- 2、调用 master.DefaultAPIResourceConfigSource 加载需要启用的 API Resource，集群中所有的 API Resource 可以在代码的 k8s.io/api 目录中看到，随着版本的迭代也会不断变化；
- 3、为 genericConfig 中的部分字段设置默认值；
- 4、调用 completedStorageFactoryConfig.New 创建 storageFactory，后面会使用 storageFactory 为每种API Resource 创建对应的 RESTStorage；

k8s.io/kubernetes/cmd/kube-apiserver/app/server.go:386

```
func buildGenericConfig(
  s *options.ServerRunOptions,
  proxyTransport *http.Transport,
) (.....) {
  // 1、为 genericConfig 设置默认值
  genericConfig = genericapiserver.NewConfig(legacyscheme.Codecs)
  genericConfig.MergedResourceConfig = master.DefaultAPIResourceConfigSource()

  if lastErr = s.GenericServerRunOptions.ApplyTo(genericConfig); lastErr != nil {
    return
  }
  ......

  genericConfig.OpenAPIConfig = genericapiserver.DefaultOpenAPIConfig(.....)
  genericConfig.OpenAPIConfig.Info.Title = "Kubernetes"
  genericConfig.LongRunningFunc = filters.BasicLongRunningRequestCheck(
    sets.NewString("watch", "proxy"),
    sets.NewString("attach", "exec", "proxy", "log", "portforward"),
  )

  kubeVersion := version.Get()
  genericConfig.Version = &kubeVersion

  storageFactoryConfig := kubeapiserver.NewStorageFactoryConfig()
  storageFactoryConfig.ApiResourceConfig = genericConfig.MergedResourceConfig
  completedStorageFactoryConfig, err := storageFactoryConfig.Complete(s.Etcd)
  if err != nil {
    lastErr = err
    return
  }
  // 初始化 storageFactory
  storageFactory, lastErr = completedStorageFactoryConfig.New()
  if lastErr != nil {
    return
  }
  if genericConfig.EgressSelector != nil {
    storageFactory.StorageConfig.Transport.EgressLookup = genericConfig.EgressSelector.Lookup
  }

  // 2、初始化 RESTOptionsGetter，后期根据其获取操作 Etcd 的句柄，同时添加 etcd 的健康检查方法
  if lastErr = s.Etcd.ApplyWithStorageFactoryTo(storageFactory, genericConfig); lastErr != nil {
    return
  }

  // 3、设置使用 protobufs 用来内部交互，并且禁用压缩功能
  genericConfig.LoopbackClientConfig.ContentConfig.ContentType = "application/vnd.kubernetes.protobuf"

  genericConfig.LoopbackClientConfig.DisableCompression = true

  // 4、创建 clientset
  kubeClientConfig := genericConfig.LoopbackClientConfig
  clientgoExternalClient, err := clientgoclientset.NewForConfig(kubeClientConfig)
  if err != nil {
    lastErr = fmt.Errorf("failed to create real external clientset: %v", err)
    return
```

```
    }
    versionedInformers = clientgoinformers.NewSharedInformerFactory(clientgoExternalClient, 10*time.Minute)

    // 5、创建认证实例，支持多种认证方式：请求 Header 认证、Auth 文件认证、CA 证书认证、Bearer token 认证、
    // ServiceAccount 认证、BootstrapToken 认证、WebhookToken 认证等
    genericConfig.Authentication.Authenticator, genericConfig.OpenAPIConfig.SecurityDefinitions, err = BuildAuthenticator(s,          clientg
    if err != nil {
        lastErr = fmt.Errorf("invalid authentication config: %v", err)
        return
    }

    // 6、创建鉴权实例，包含：Node、RBAC、Webhook、ABAC、AlwaysAllow、AlwaysDeny
    genericConfig.Authorization.Authorizer, genericConfig.RuleResolver, err = BuildAuthorizer(s, versionedInformers)
    ......

    serviceResolver = buildServiceResolver(s.EnableAggregatorRouting, genericConfig.LoopbackClientConfig.Host, versionedInformers)

    authInfoResolverWrapper := webhook.NewDefaultAuthenticationInfoResolverWrapper(proxyTransport, genericConfig.LoopbackClientCo

    // 7、审计插件的初始化
    lastErr = s.Audit.ApplyTo(......)
    if lastErr != nil {
        return
    }

    // 8、准入插件的初始化
    pluginInitializers, admissionPostStartHook, err = admissionConfig.New(proxyTransport, serviceResolver)
    if err != nil {
        lastErr = fmt.Errorf("failed to create admission plugin initializer: %v", err)
        return
    }
    err = s.Admission.ApplyTo(.....)
    if err != nil {
        lastErr = fmt.Errorf("failed to initialize admission: %v", err)
    }

    return
}
```

以上主要分析 KubeAPIServerConfig 的初始化，其他两个 server config 的初始化暂且不详细分析，下面接着继续分析 server 的初始化。

createAPIExtensionsServer

APIExtensionsServer 是最先被初始化的，在 createAPIExtensionsServer 中调用 apiextensionsConfig.Complete().New 来完成 server 的初始化，其主要逻辑为：

- 1、首先调用 c.GenericConfig.New 按照 go-restful 的模式初始化 Container，在 c.GenericConfig.New 中会调用 NewAPIServerHandler 初始化 handler，APIServerHandler 包含了 API Server 使用的多种http.Handler 类型，包括 go-restful 以及 non-go-restful ，以及在以上两者之间选择的 Director 对象， go-restful 用于处理已经注册的 handler， non-go-restful 用来处理不存在的 handler，API URI 处理的选择过程为： FullHandlerChain-> Director ->{GoRestfulContainer， NonGoRestfulMux} 。在 c.GenericConfig.New 中还会调用 installAPI 来添加包括 / 、 /debug/* 、 /metrics 、 /version 等路由信息。三种 server 在初始化时首先都会调用 c.GenericConfig.New 来初始化一个 genericServer，然后进行 API 的注册；
- 2、调用 s.GenericAPIServer.InstallAPIGroup 在路由中注册 API Resources，此方法的调用链非常深，主要是为了将需要暴露的 API Resource 注册到 server 中，以便能通过 http 接口进行 resource 的 REST 操作，其他几种 server 在初始化时也都会执行对应的 InstallAPI ；

```
38
39      ......
40      establishingController := establish.NewEstablishingController(s.Informers.Apiextensions().InternalVersion().                CustomResourceDe
41      crdHandler, err := NewCustomResourceDefinitionHandler(......)
42      if err != nil {
43          return nil, err
44      }
45      s.GenericAPIServer.Handler.NonGoRestfulMux.Handle("/apis", crdHandler)
46      s.GenericAPIServer.Handler.NonGoRestfulMux.HandlePrefix("/apis/", crdHandler)
47
48      crdController := NewDiscoveryController(s.Informers.Apiextensions().InternalVersion().CustomResourceDefinitions(),          versionDisc
49      namingController := status.NewNamingConditionController(s.Informers.Apiextensions().InternalVersion().CustomResourceDefinitions(),
50      nonStructuralSchemaController := nonstructuralschema.NewConditionController(s.Informers.Apiextensions().InternalVersion().      Cust
51      apiApprovalController := apiapproval.NewKubernetesAPIApprovalPolicyConformantConditionController(s.Informers.Apiextensions().   I
52      finalizingController := finalizer.NewCRDFinalizer(
53          s.Informers.Apiextensions().InternalVersion().CustomResourceDefinitions(),
54          crdClient.Apiextensions(),
55          crdHandler,
56      )
57      var openapiController *openapicontroller.Controller
58      if utilfeature.DefaultFeatureGate.Enabled(apiextensionsfeatures.CustomResourcePublishOpenAPI) {
59          openapiController = openapicontroller.NewController(s.Informers.Apiextensions().InternalVersion().CustomResourceDefinitions())
60      }
61
62      // 5、将 informer 以及 controller 添加到 PostStartHook 中
63      s.GenericAPIServer.AddPostStartHookOrDie("start-apiextensions-informers", func(context genericapiserver.PostStartHookContext) error
64          s.Informers.Start(context.StopCh)
65          return nil
66      })
67      s.GenericAPIServer.AddPostStartHookOrDie("start-apiextensions-controllers", func(context genericapiserver.PostStartHookContext) erro
68          ......
69          go crdController.Run(context.StopCh)
70          go namingController.Run(context.StopCh)
71          go establishingController.Run(context.StopCh)
72          go nonStructuralSchemaController.Run(5, context.StopCh)
73          go apiApprovalController.Run(5, context.StopCh)
74          go finalizingController.Run(5, context.StopCh)
75          return nil
76      })
77
78      s.GenericAPIServer.AddPostStartHookOrDie("crd-informer-synced", func(context genericapiserver.PostStartHookContext) error {
79          return wait.PollImmediateUntil(100*time.Millisecond, func() (bool, error) {
80              return s.Informers.Apiextensions().InternalVersion().CustomResourceDefinitions().Informer().HasSynced(), nil
81          }, context.StopCh)
82      })
83
84      return s, nil
85  }
```

以上是 APIExtensionsServer 的初始化流程，其中最核心方法是 s.GenericAPIServer.InstallAPIGroup ，也就是 API 的注册过程，三种 server 中 API 的注册过程都是其核心。

CreateKubeAPIServer

本节继续分析 KubeAPIServer 的初始化，在 CreateKubeAPIServer 中调用了 kubeAPIServerConfig.Complete().New 来完成相关的初始化操作。

kubeAPIServerConfig.Complete().New

主要逻辑为：

- 1、调用 c.GenericConfig.New 初始化 GenericAPIServer，其主要实现在上文已经分析过；
- 2、判断是否支持 logs 相关的路由，如果支持，则添加 /logs 路由；
- 3、调用 m.InstallLegacyAPI 将核心 API Resource 添加到路由中，对应到 apiserver 就是以 /api 开头的 resource；

- 4、调用 m.InstallAPIs 将扩展的 API Resource 添加到路由中，在 apiserver 中即是以 /apis 开头的 resource；

k8s.io/kubernetes/cmd/kube-apiserver/app/server.go:214

```
func CreateKubeAPIServer(.....) (*master.Master, error) {
    kubeAPIServer, err := kubeAPIServerConfig.Complete().New(delegateAPIServer)
    if err != nil {
        return nil, err
    }

    kubeAPIServer.GenericAPIServer.AddPostStartHookOrDie("start-kube-apiserver-admission-initializer", admissionPostStartHook)

    return kubeAPIServer, nil
}
```

k8s.io/kubernetes/pkg/master/master.go:325

```
func (c completedConfig) New(delegationTarget genericapiserver.DelegationTarget) (*Master, error) {
    ......
    // 1、初始化 GenericAPIServer
    s, err := c.GenericConfig.New("kube-apiserver", delegationTarget)
    if err != nil {
        return nil, err
    }

    // 2、注册 logs 相关的路由
    if c.ExtraConfig.EnableLogsSupport {
        routes.Logs{}.Install(s.Handler.GoRestfulContainer)
    }

    m := &Master{
        GenericAPIServer: s,
    }

    // 3、安装 LegacyAPI
    if c.ExtraConfig.APIResourceConfigSource.VersionEnabled(apiv1.SchemeGroupVersion) {
        legacyRESTStorageProvider := corerest.LegacyRESTStorageProvider{
            StorageFactory:          c.ExtraConfig.StorageFactory,
            ProxyTransport:          c.ExtraConfig.ProxyTransport,
            ......
        }
        if err := m.InstallLegacyAPI(&c, c.GenericConfig.RESTOptionsGetter, legacyRESTStorageProvider); err != nil {
            return nil, err
        }
    }
    restStorageProviders := []RESTStorageProvider{
        auditregistrationrest.RESTStorageProvider{},
        authenticationrest.RESTStorageProvider{Authenticator: c.GenericConfig.Authentication.Authenticator, APIAudiences: c.GenericConfig.
        ......
    }
    // 4、安装 APIs
    if err := m.InstallAPIs(c.ExtraConfig.APIResourceConfigSource, c.GenericConfig.RESTOptionsGetter, restStorageProviders...); err != nil {
        return nil, err
    }

    if c.ExtraConfig.Tunneler != nil {
        m.installTunneler(c.ExtraConfig.Tunneler, corev1client.NewForConfigOrDie(c.GenericConfig.LoopbackClientConfig).Nodes())
    }

    m.GenericAPIServer.AddPostStartHookOrDie("ca-registration", c.ExtraConfig.ClientCARegistrationHook.PostStartHook)

    return m, nil
}
```

m.InstallLegacyAPI

此方法的主要功能是将 core API 注册到路由中，是 apiserver 初始化流程中最核心的方法之一，不过其调用链非常深，下面会进行深入分析。将 API 注册到路由其最终的目的就是对外提供 RESTful API 来操作对应 resource，注册 API 主要分为两步，第一步是为 API 中的每个 resource 初始化 RESTStorage 以此操作后端存储中数据的变更，第二步是为每个 resource 根据其 verbs 构建对应的路由。 m.InstallLegacyAPI 的主要逻辑为：

- 1、调用 legacyRESTStorageProvider.NewLegacyRESTStorage 为 LegacyAPI 中各个资源创建 RESTStorage，RESTStorage 的目的是将每种资源的访问路径及其后端存储的操作对应起来；
- 2、初始化 bootstrap-controller ，并将其加入到 PostStartHook 中， bootstrap-controller 是 apiserver 中的一个 controller，主要功能是创建系统所需要的一些 namespace 以及创建 kubernetes service 并定期触发对应的 sync 操作，apiserver 在启动后会通过调用 PostStartHook 来启动 bootstrap-controller ；
- 3、在为资源创建完 RESTStorage 后，调用 m.GenericAPIServer.InstallLegacyAPIGroup 为 APIGroup 注册路由信息， InstallLegacyAPIGroup 方法的调用链非常深，主要为 InstallLegacyAPIGroup--> installAPIResources --> InstallREST --> Install --> registerResourceHandlers ，最终核心的路由构造在 registerResourceHandlers 方法内，该方法比较复杂，其主要功能是通过上一步骤构造的 REST Storage 判断该资源可以执行哪些操作（如 create、update等），将其对应的操作存入到 action 中，每一个 action 对应一个标准的 REST 操作，如 create 对应的 action 操作为 POST、update 对应的 action 操作为PUT。最终根据 actions 数组依次遍历，对每一个操作添加一个 handler 方法，注册到 route 中去，再将 route 注册到 webservice 中去，webservice 最终会注册到 container 中，遵循 go-restful 的设计模式；

关于 legacyRESTStorageProvider.NewLegacyRESTStorage 以及 m.GenericAPIServer.InstallLegacyAPIGroup 方法的详细说明在后文中会继续进行讲解。

k8s.io/kubernetes/pkg/master/master.go:406

```
1  func (m *Master) InstallLegacyAPI(.....) error {
2      legacyRESTStorage, apiGroupInfo, err := legacyRESTStorageProvider.NewLegacyRESTStorage(restOptionsGetter)
3      if err != nil {
4          return fmt.Errorf("Error building core storage: %v", err)
5      }
6
7      controllerName := "bootstrap-controller"
8      coreClient := corev1client.NewForConfigOrDie(c.GenericConfig.LoopbackClientConfig)
9      bootstrapController := c.NewBootstrapController(legacyRESTStorage, coreClient, coreClient, coreClient, coreClient.RESTClient())
10     m.GenericAPIServer.AddPostStartHookOrDie(controllerName, bootstrapController.PostStartHook)
11     m.GenericAPIServer.AddPreShutdownHookOrDie(controllerName, bootstrapController.PreShutdownHook)
12
13     if err := m.GenericAPIServer.InstallLegacyAPIGroup(genericapiserver.DefaultLegacyAPIPrefix, &apiGroupInfo); err != nil {
14         return fmt.Errorf("Error in registering group versions: %v", err)
15     }
16     return nil
17 }
```

InstallAPIs 与 InstallLegacyAPI 的主要流程是类似的，限于篇幅此处不再深入分析。

createAggregatorServer

AggregatorServer 主要用于自定义的聚合控制器的，使 CRD 能够自动注册到集群中。

主要逻辑为：

- 1、调用 aggregatorConfig.Complete().NewWithDelegate 创建 aggregatorServer；
- 2、初始化 crdRegistrationController 和 autoRegistrationController ， crdRegistrationController 负责注册 CRD， autoRegistrationController 负责将 CRD 对应的 APIServices 自动注册到 apiserver 中，CRD 创建后可通过 $ kubectl get apiservices 查看是否注册到 apiservices 中；
- 3、将 autoRegistrationController 和 crdRegistrationController 加入到 PostStartHook 中；

k8s.io/kubernetes/cmd/kube-apiserver/app/aggregator.go:124

```
1  func createAggregatorServer(.....) (*aggregatorapiserver.APIAggregator, error) {
```

```
    // 1、初始化 aggregatorServer
    aggregatorServer, err := aggregatorConfig.Complete().NewWithDelegate(delegateAPIServer)
    if err != nil {
        return nil, err
    }

    // 2、初始化 auto-registration controller
    apiRegistrationClient, err := apiregistrationclient.NewForConfig(aggregatorConfig.GenericConfig.LoopbackClientConfig)
    if err != nil {
        return nil, err
    }
    autoRegistrationController := autoregister.NewAutoRegisterController(.....)
    apiServices := apiServicesToRegister(delegateAPIServer, autoRegistrationController)
    crdRegistrationController := crdregistration.NewCRDRegistrationController(.....)
    err = aggregatorServer.GenericAPIServer.AddPostStartHook("kube-apiserver-autoregistration", func(context genericapiserver.PostStartH
        go crdRegistrationController.Run(5, context.StopCh)
        go func() {
            if aggregatorConfig.GenericConfig.MergedResourceConfig.AnyVersionForGroupEnabled("apiextensions.k8s.io") {
                crdRegistrationController.WaitForInitialSync()
            }
            autoRegistrationController.Run(5, context.StopCh)
        }()
        return nil
    })
    if err != nil {
        return nil, err
    }

    err = aggregatorServer.GenericAPIServer.AddBootSequenceHealthChecks(
        makeAPIServiceAvailableHealthCheck(
            "autoregister-completion",
            apiServices,
            aggregatorServer.APIRegistrationInformers.Apiregistration().V1().APIServices(),
        ),
    )
    if err != nil {
        return nil, err
    }

    return aggregatorServer, nil
}
```

aggregatorConfig.Complete().NewWithDelegate

aggregatorConfig.Complete().NewWithDelegate 是初始化 aggregatorServer 的方法，主要逻辑为：

- 1、调用 c.GenericConfig.New 初始化 GenericAPIServer，其内部的主要功能在上文已经分析过；
- 2、调用 apiservicerest.NewRESTStorage 为 APIServices 资源创建 RESTStorage，RESTStorage 的目的是将每种资源的访问路径及其后端存储的操作对应起来；
- 3、调用 s.GenericAPIServer.InstallAPIGroup 为 APIGroup 注册路由信息；

k8s.io/kubernetes/staging/src/k8s.io/kube-aggregator/pkg/apiserver/apiserver.go:158

```
func (c completedConfig) NewWithDelegate(delegationTarget genericapiserver.DelegationTarget) (*APIAggregator, error) {
    openAPIConfig := c.GenericConfig.OpenAPIConfig
    c.GenericConfig.OpenAPIConfig = nil
    // 1、初始化 genericServer
    genericServer, err := c.GenericConfig.New("kube-aggregator", delegationTarget)
    if err != nil {
        return nil, err
    }

    apiregistrationClient, err := clientset.NewForConfig(c.GenericConfig.LoopbackClientConfig)
```

```
    if err != nil {
        return nil, err
    }
    informerFactory := informers.NewSharedInformerFactory(
        apiregistrationClient,
        5*time.Minute,
    )
    s := &APIAggregator{
        GenericAPIServer: genericServer,
        delegateHandler: delegationTarget.UnprotectedHandler(),
        ......
    }

    // 2、为 API 注册路由
    apiGroupInfo := apiservicerest.NewRESTStorage(c.GenericConfig.MergedResourceConfig, c.GenericConfig.RESTOptionsGetter)
    if err := s.GenericAPIServer.InstallAPIGroup(&apiGroupInfo); err != nil {
        return nil, err
    }

    // 3、初始化 apiserviceRegistrationController、availableController
    apisHandler := &apisHandler{
        codecs: aggregatorscheme.Codecs,
        lister: s.lister,
    }
    s.GenericAPIServer.Handler.NonGoRestfulMux.Handle("/apis", apisHandler)
    s.GenericAPIServer.Handler.NonGoRestfulMux.UnlistedHandle("/apis/", apisHandler)
    apiserviceRegistrationController := NewAPIServiceRegistrationController(informerFactory.Apiregistration().V1().APIServices(), s)
    availableController, err := statuscontrollers.NewAvailableConditionController(
      ......
    )
    if err != nil {
        return nil, err
    }

    // 4、添加 PostStartHook
    s.GenericAPIServer.AddPostStartHookOrDie("start-kube-aggregator-informers", func(context genericapiserver.PostStartHookContext) err
        informerFactory.Start(context.StopCh)
        c.GenericConfig.SharedInformerFactory.Start(context.StopCh)
        return nil
    })
    s.GenericAPIServer.AddPostStartHookOrDie("apiservice-registration-controller", func(context genericapiserver.PostStartHookContext)
        go apiserviceRegistrationController.Run(context.StopCh)
        return nil
    })
    s.GenericAPIServer.AddPostStartHookOrDie("apiservice-status-available-controller", func(context genericapiserver.PostStartHookConte
        go availableController.Run(5, context.StopCh)
        return nil
    })

    return s, nil
}
```

以上是对 AggregatorServer 初始化流程的分析，可以看出，在创建 APIExtensionsServer、KubeAPIServer 以及 AggregatorServer 时，其模式都是类似的，首先调用 c.GenericConfig.New 按照 go-restful 的模式初始化 Container，然后为 server 中需要注册的资源创建 RESTStorage，最后将 resource 的 APIGroup 信息注册到路由中。

至此，CreateServerChain 中流程已经分析完，其中的调用链如下所示：

```
                |--> CreateNodeDialer
                |
                |--> CreateKubeAPIServerConfig
                |
CreateServerChain --|--> createAPIExtensionsConfig
                |
```

```
                |                                          |--> c.GenericConfig.New
                |--> createAPIExtensionsServer --> apiextensionsConfig.Complete().New --|
                |                                          |--> s.GenericAPIServer.InstallAPIGroup
                |
                |                                  |--> c.GenericConfig.New --> legacyRESTStorageProvider.NewLegacyRESTStorage
                |                                  |
                |--> CreateKubeAPIServer --> kubeAPIServerConfig.Complete().New --|--> m.InstallLegacyAPI
                |                                  |
                |                                  |--> m.InstallAPIs
                |
                |
                |--> createAggregatorConfig
                |
                |                                          |--> c.GenericConfig.New
                |                                          |
                |--> createAggregatorServer --> aggregatorConfig.Complete().NewWithDelegate --|--> apiservicerest.NewRESTStorage
                                                           |
                                                           |--> s.GenericAPIServer.InstallAPIGroup
```

prepared.Run

在 Run 方法中首先调用 CreateServerChain 完成各 server 的初始化，然后调用 server.PrepareRun 完成服务启动前的准备工作，最后调用 prepared.Run 方法来启动安全的 http server。 server.PrepareRun 主要完成了健康检查、存活检查和 OpenAPI 路由的注册工作，下面继续分析 prepared.Run 的流程，在 prepared.Run 中主要调用 s.NonBlockingRun 来完成启动工作。

k8s.io/kubernetes/staging/src/k8s.io/kube-aggregator/pkg/apiserver/apiserver.go:269

```
func (s preparedAPIAggregator) Run(stopCh <-chan struct{}) error {
  return s.runnable.Run(stopCh)
}
```

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/server/genericapiserver.go:316

```
func (s preparedGenericAPIServer) Run(stopCh <-chan struct{}) error {
  delayedStopCh := make(chan struct{})

  go func() {
    defer close(delayedStopCh)
    <-stopCh

    time.Sleep(s.ShutdownDelayDuration)
  }()

  // 调用 s.NonBlockingRun 完成启动流程
  err := s.NonBlockingRun(delayedStopCh)
  if err != nil {
    return err
  }

  // 当收到退出信号后完成一些收尾工作
  <-stopCh
  err = s.RunPreShutdownHooks()
  if err != nil {
    return err
  }

  <-delayedStopCh
  s.HandlerChainWaitGroup.Wait()
  return nil
}
```

s.NonBlockingRun

s.NonBlockingRun 的主要逻辑为：

- 1、判断是否要启动审计日志服务；
- 2、调用 s.SecureServingInfo.Serve 配置并启动 https server；
- 3、执行 postStartHooks；
- 4、向 systemd 发送 ready 信号；

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/server/genericapiserver.go:351

```
func (s preparedGenericAPIServer) NonBlockingRun(stopCh <-chan struct{}) error {
    auditStopCh := make(chan struct{})

    // 1、判断是否要启动审计日志
    if s.AuditBackend != nil {
        if err := s.AuditBackend.Run(auditStopCh); err != nil {
            return fmt.Errorf("failed to run the audit backend: %v", err)
        }
    }

    // 2、启动 https server
    internalStopCh := make(chan struct{})
    var stoppedCh <-chan struct{}
    if s.SecureServingInfo != nil && s.Handler != nil {
        var err error
        stoppedCh, err = s.SecureServingInfo.Serve(s.Handler, s.ShutdownTimeout, internalStopCh)
        if err != nil {
            close(internalStopCh)
            close(auditStopCh)
            return err
        }
    }

    go func() {
        <-stopCh
        close(s.readinessStopCh)
        close(internalStopCh)
        if stoppedCh != nil {
            <-stoppedCh
        }
        s.HandlerChainWaitGroup.Wait()
        close(auditStopCh)
    }()

    // 3、执行 postStartHooks
    s.RunPostStartHooks(stopCh)

    // 4、向 systemd 发送 ready 信号
    if _, err := systemd.SdNotify(true, "READY=1\n"); err != nil {
        klog.Errorf("Unable to send systemd daemon successful start message: %v\n", err)
    }

    return nil
}
```

以上就是 server 的初始化以及启动流程过程的分析，上文已经提到各 server 初始化过程中最重要的就是 API Resource RESTStorage 的初始化以及路由的注册，由于该过程比较复杂，下文会单独进行讲述。

storageFactory 的构建

上文已经提到过，apiserver 最终实现的 handler 对应的后端数据是以 **Store** 的结构保存的，这里以 /api 开头的路由举例，通过 NewLegacyRESTStorage 方法创建各个资源的**RESTStorage**。RESTStorage 是一个结构体，具体的定义在 k8s.io/apiserver/pkg/registry/generic/registry/store.go 下，结构体内主要包含

v1beta1⇔v1alpha1），因此其使用了一个特殊的 internal version， internal version 作为一个通用的 version 会包含所有 version 的字段，它具有所有 version 的功能。 Decoder 会首先把 creater object 转换到 internal version，然后将其转换为 storage version， storage version 是在 etcd 中存储时的另一个 version。

在解码时，首先从 HTTP path 中获取期待的 version，然后使用 scheme 以正确的 version 创建一个与之匹配的空对象，并使用 JSON 或 protobuf 解码器进行转换，在转换的第一步中，如果用户省略了某些字段，Decoder 会把其设置为默认值。

Admission

在解码完成后，需要通过验证集群的全局约束来检查是否可以创建或更新对象，并根据集群配置设置默认值。在 k8s.io/kubernetes/plugin/pkg/admission 目录下可以看到 kube-apiserver 可以使用的所有全局约束插件，kube-apiserver 在启动时通过设置 --enable-admission-plugins 参数来开启需要使用的插件，通过 ValidatingAdmissionWebhook 或 MutatingAdmissionWebhook 添加的插件也都会在此处进行工作。

Validation

主要检查 object 中字段的合法性。

在 handler 中执行完以上操作后最后会执行与 etcd 相关的操作，POST 操作会将数据写入到 etcd 中，以上在 handler 中的主要处理流程如下所示：

```
1    v1beta1 ⇒ internal ⇒  |  ⇒    |  ⇒ v1 ⇒ json/yaml ⇒ etcd
2                admission   validation
```

kube-apiserver 中的组件

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

- **Aggregator**：暴露的功能类似于一个七层负载均衡，将来自用户的请求拦截转发给其他服务器，并且负责整个 APIServer 的 Discovery 功能；
- **KubeAPIServer**：负责对请求的一些通用处理，认证、鉴权等，以及处理各个内建资源的 REST 服务；

- **APIExtensionServer**：主要处理 CustomResourceDefinition（CRD）和 CustomResource（CR）的 REST 请求，也是 Delegation 的最后一环，如果对应 CR 不能被处理的话则会返回 404。

Aggregator 和 APIExtensionsServer 对应两种主要扩展 APIServer 资源的方式，即分别是 AA 和 CRD。

Aggregator

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

Aggregator 除了处理资源请求外还包含几个 controller：

- 1、 apiserviceRegistrationController： 负责 APIServices 中资源的注册与删除；
- 2、 availableConditionController： 维护 APIServices 的可用状态，包括其引用 Service 是否可用等；

NewFunc 返回特定资源信息、 NewListFunc 返回特定资源列表、 CreateStrategy 特定资源创建时的策略、 UpdateStrategy 更新时的策略以及 DeleteStrategy 删除时的策略等重要方法。在 NewLegacyRESTStorage 内部，可以看到创建了多种资源的 RESTStorage。

NewLegacyRESTStorage 的调用链为 CreateKubeAPIServer --> kubeAPIServerConfig.Complete().New --> m.InstallLegacyAPI --> legacyRESTStorageProvider.NewLegacyRESTStorage 。

NewLegacyRESTStorage

一个 API Group 下的资源都有其 REST 实现， k8s.io/kubernetes/pkg/registry 下所有的 Group 都有一个rest目录，存储的就是对应资源的 RESTStorage。在 NewLegacyRESTStorage 方法中，通过 NewREST 或者 NewStorage 会生成各种资源对应的 Storage，此处以 pod 为例进行说明。

k8s.io/kubernetes/pkg/registry/core/rest/storage_core.go:102

```
func (c LegacyRESTStorageProvider) NewLegacyRESTStorage(restOptionsGetter generic.RESTOptionsGetter) (LegacyRESTStorage, generica
  apiGroupInfo := genericapiserver.APIGroupInfo{
    PrioritizedVersions:        legacyscheme.Scheme.PrioritizedVersionsForGroup(""),
    VersionedResourcesStorageMap: map[string]map[string]rest.Storage{},
    Scheme:                     legacyscheme.Scheme,
    ParameterCodec:             legacyscheme.ParameterCodec,
    NegotiatedSerializer:       legacyscheme.Codecs,
  }

  var podDisruptionClient policyclient.PodDisruptionBudgetsGetter
  if policyGroupVersion := (schema.GroupVersion{Group: "policy", Version: "v1beta1"}); legacyscheme.Scheme.                IsVersionReg
    var err error
    podDisruptionClient, err = policyclient.NewForConfig(c.LoopbackClientConfig)
    if err != nil {
      return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, err
    }
  }
  // 1、LegacyAPI 下的 resource RESTStorage 的初始化
  restStorage := LegacyRESTStorage{}

  podTemplateStorage, err := podtemplatestore.NewREST(restOptionsGetter)
  if err != nil {
    return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, err
  }
  eventStorage, err := eventstore.NewREST(restOptionsGetter, uint64(c.EventTTL.Seconds()))
  if err != nil {
    return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, err
  }
  limitRangeStorage, err := limitrangestore.NewREST(restOptionsGetter)
  if err != nil {
    return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, err
  }

  ......

  endpointsStorage, err := endpointsstore.NewREST(restOptionsGetter)
  if err != nil {
    return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, err
  }

  nodeStorage, err := nodestore.NewStorage(restOptionsGetter, c.KubeletClientConfig, c.ProxyTransport)
  if err != nil {
    return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, err
  }

  // 2、pod RESTStorage 的初始化
  podStorage, err := podstore.NewStorage(......)
  if err != nil {
    return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, err
  }
```

```
    ......

    serviceClusterIPAllocator, err := ipallocator.NewAllocatorCIDRRange(&serviceClusterIPRange, func(max int, rangeSpec string) (allocator. I
      ......
    })
    if err != nil {
      return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, fmt.Errorf("cannot create cluster IP allocator: %v", err)
    }
    restStorage.ServiceClusterIPAllocator = serviceClusterIPRegistry

    var secondaryServiceClusterIPAllocator ipallocator.Interface
    if utilfeature.DefaultFeatureGate.Enabled(features.IPv6DualStack) && c.SecondaryServiceIPRange.IP != nil {
      ......
    }

    var serviceNodePortRegistry rangeallocation.RangeRegistry
    serviceNodePortAllocator, err := portallocator.NewPortAllocatorCustom(c.ServiceNodePortRange, func(max int, rangeSpec string)     (allo
      ......
    })
    if err != nil {
      return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, fmt.Errorf("cannot create cluster port allocator: %v", err)
    }
    restStorage.ServiceNodePortAllocator = serviceNodePortRegistry

    controllerStorage, err := controllerstore.NewStorage(restOptionsGetter)
    if err != nil {
      return LegacyRESTStorage{}, genericapiserver.APIGroupInfo{}, err
    }

    serviceRest, serviceRestProxy := servicestore.NewREST(......)

    // 3、restStorageMap 保存 resource http path 与 RESTStorage 对应关系
    restStorageMap := map[string]rest.Storage{
      "pods":           podStorage.Pod,
      "pods/attach":     podStorage.Attach,
      "pods/status":     podStorage.Status,
      "pods/log":       podStorage.Log,
      "pods/exec":       podStorage.Exec,
      "pods/portforward": podStorage.PortForward,
      "pods/proxy":      podStorage.Proxy,
      ......
      "componentStatuses": componentstatus.NewStorage(componentStatusStorage{c.StorageFactory}.serversToValidate),
    }
    ......
  }
```

podstore.NewStorage

podstore.NewStorage 是为 pod 生成 storage 的方法，该方法主要功能是为 pod 创建后端存储最终返回一个 RESTStorage 对象，其中调用 store.CompleteWithOptions 来创建后端存储的。

k8s.io/kubernetes/pkg/registry/core/pod/storage/storage.go:71

```
func NewStorage(......) (PodStorage, error) {
  store := &genericregistry.Store{
    NewFunc:                  func() runtime.Object { return &api.Pod{} },
    NewListFunc:              func() runtime.Object { return &api.PodList{} },
    ......
  }
  options := &generic.StoreOptions{
    RESTOptions: optsGetter,
    AttrFunc:    pod.GetAttrs,
    TriggerFunc: map[string]storage.IndexerFunc{"spec.nodeName": pod.NodeNameTriggerFunc},
  }
```

```
12
13      // 调用 store.CompleteWithOptions
14      if err := store.CompleteWithOptions(options); err != nil {
15          return PodStorage{}, err
16      }
17      statusStore := *store
18      statusStore.UpdateStrategy = pod.StatusStrategy
19      ephemeralContainersStore := *store
20      ephemeralContainersStore.UpdateStrategy = pod.EphemeralContainersStrategy
21
22      bindingREST := &BindingREST{store: store}
23
24      // PodStorage 对象
25      return PodStorage{
26          Pod:              &REST{store, proxyTransport},
27          Binding:          &BindingREST{store: store},
28          LegacyBinding:    &LegacyBindingREST{bindingREST},
29          Eviction:         newEvictionStorage(store, podDisruptionBudgetClient),
30          Status:           &StatusREST{store: &statusStore},
31          EphemeralContainers: &EphemeralContainersREST{store: &ephemeralContainersStore},
32          Log:              &podrest.LogREST{Store: store, KubeletConn: k},
33          Proxy:            &podrest.ProxyREST{Store: store, ProxyTransport: proxyTransport},
34          Exec:             &podrest.ExecREST{Store: store, KubeletConn: k},
35          Attach:           &podrest.AttachREST{Store: store, KubeletConn: k},
36          PortForward:      &podrest.PortForwardREST{Store: store, KubeletConn: k},
37      }, nil
38  }
```

可以看到最终返回的对象里对 pod 的不同操作都是一个 REST 对象，REST 中自动集成了 genericregistry.Store 对象，而 store.CompleteWithOptions 方法就是对 genericregistry.Store 对象中存储实例就行初始化的。

```
1   type REST struct {
2       *genericregistry.Store
3       proxyTransport http.RoundTripper
4   }
5
6   type BindingREST struct {
7       store *genericregistry.Store
8   }
9   ......
```

store.CompleteWithOptions

store.CompleteWithOptions 主要功能是为 store 中的配置设置一些默认的值以及根据提供的 options 更新 store，其中最主要的就是初始化 store 的后端存储实例。

在 CompleteWithOptions 方法内，调用了 options.RESTOptions.GetRESTOptions 方法，其最终返回 generic.RESTOptions 对象，generic.RESTOptions 对象中包含对 etcd 初始化的一些配置、数据序列化方法以及对 etcd 操作的 storage.Interface 对象。其会依次调用 StorageWithCacher-->NewRawStorage-->Create 方法创建最终依赖的后端存储。

```
50          e.Storage.Codec = opts.StorageConfig.Codec
51          var err error
52          e.Storage.Storage, e.DestroyFunc, err = opts.Decorator(
53              opts.StorageConfig,
54              prefix,
55              keyFunc,
56              e.NewFunc,
57              e.NewListFunc,
58              attrFunc,
59              options.TriggerFunc,
60          )
61          if err != nil {
62              return err
63          }
64          e.StorageVersioner = opts.StorageConfig.EncodeVersioner
65
66          if opts.CountMetricPollPeriod > 0 {
67              stopFunc := e.startObservingCount(opts.CountMetricPollPeriod)
68              previousDestroy := e.DestroyFunc
69              e.DestroyFunc = func() {
70                  stopFunc()
71                  if previousDestroy != nil {
72                      previousDestroy()
73                  }
74              }
75          }
76      }
77
78      return nil
79  }
```

options.RESTOptions 是一个 interface，想要找到其 GetRESTOptions 方法的实现必须知道 options.RESTOptions 初始化时对应的实例，其初始化是在 CreateKubeAPIServerConfig --> buildGenericConfig --> s.Etcd.ApplyWithStorageFactoryTo 方法中进行初始化的， RESTOptions 对应的实例为 StorageFactoryRestOptionsFactory ，所以 PodStorage 初始时构建的 store 对象中 genericserver.Config.RESTOptionsGetter 实际的对象类型为 StorageFactoryRestOptionsFactory ，其 GetRESTOptions 方法如下所示：

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/server/options/etcd.go:253

```
1   func (f *StorageFactoryRestOptionsFactory) GetRESTOptions(resource schema.GroupResource) (generic.RESTOptions, error) {
2       storageConfig, err := f.StorageFactory.NewConfig(resource)
3       if err != nil {
4           return generic.RESTOptions{}, fmt.Errorf("unable to find storage destination for %v, due to %v", resource, err.Error())
5       }
6
7       ret := generic.RESTOptions{
8           StorageConfig:           storageConfig,
9           Decorator:               generic.UndecoratedStorage,
10          DeleteCollectionWorkers: f.Options.DeleteCollectionWorkers,
11          EnableGarbageCollection: f.Options.EnableGarbageCollection,
12          ResourcePrefix:          f.StorageFactory.ResourcePrefix(resource),
13          CountMetricPollPeriod:   f.Options.StorageConfig.CountMetricPollPeriod,
14      }
15      if f.Options.EnableWatchCache {
16          sizes, err := ParseWatchCacheSizes(f.Options.WatchCacheSizes)
17          if err != nil {
18              return generic.RESTOptions{}, err
19          }
20          cacheSize, ok := sizes[resource]
21          if !ok {
22              cacheSize = f.Options.DefaultWatchCacheSize
23          }
24          // 调用 generic.StorageDecorator
25          ret.Decorator = genericregistry.StorageWithCacher(cacheSize)
26      }
```

```
27
28      return ret, nil
29  }
```

在 genericregistry.StorageWithCacher 中又调用了不同的方法最终会调用 factory.Create 来初始化存储实例，其调用链为：genericregistry.StorageWithCacher --> generic.NewRawStorage --> factory.Create 。

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/storage/storagebackend/factory/factory.go:30

```
1   func Create(c storagebackend.Config) (storage.Interface, DestroyFunc, error) {
2       switch c.Type {
3       case "etcd2":
4           return nil, nil, fmt.Errorf("%v is no longer a supported storage backend", c.Type)
5       // 目前 k8s 只支持使用 etcd v3
6       case storagebackend.StorageTypeUnset, storagebackend.StorageTypeETCD3:
7           return newETCD3Storage(c)
8       default:
9           return nil, nil, fmt.Errorf("unknown storage type: %s", c.Type)
10      }
11  }
```

newETCD3Storage

在 newETCD3Storage 中，首先通过调用 newETCD3Client 创建 etcd 的 client，client 的创建最终是通过 etcd 官方提供的客户端工具 clientv3 进行创建的。

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/storage/storagebackend/factory/etcd3.go:209

```
1   func newETCD3Storage(c storagebackend.Config) (storage.Interface, DestroyFunc, error) {
2       stopCompactor, err := startCompactorOnce(c.Transport, c.CompactionInterval)
3       if err != nil {
4           return nil, nil, err
5       }
6
7       client, err := newETCD3Client(c.Transport)
8       if err != nil {
9           stopCompactor()
10          return nil, nil, err
11      }
12
13      var once sync.Once
14      destroyFunc := func() {
15          once.Do(func() {
16              stopCompactor()
17              client.Close()
18          })
19      }
20      transformer := c.Transformer
21      if transformer == nil {
22          transformer = value.IdentityTransformer
23      }
24      return etcd3.New(client, c.Codec, c.Prefix, transformer, c.Paging), destroyFunc, nil
25  }
```

至此对于 pod resource 中 store 的构建基本分析完成，不同 resource 对应一个 REST 对象，其中又引用了 genericregistry.Store 对象，最终是对 genericregistry.Store 的初始化。在分析完 store 的初始化后还有一个重要的步骤就是路由的注册，路由注册主要的流程是为 resource 根据不同 verbs 构建 http path 以及将 path 与对应 handler 进行绑定。

路由注册

上文 RESTStorage 的构建对应的是 InstallLegacyAPI 中的 legacyRESTStorageProvider.NewLegacyRESTStorage 方法，下面继续分析 InstallLegacyAPI 中的 m.GenericAPIServer.InstallLegacyAPIGroup 方法的实现。

k8s.io/kubernetes/pkg/master/master.go:406

```
func (m *Master) InstallLegacyAPI(.....) error {
    legacyRESTStorage, apiGroupInfo, err := legacyRESTStorageProvider.NewLegacyRESTStorage(restOptionsGetter)
    if err != nil {
        return fmt.Errorf("Error building core storage: %v", err)
    }
    .....

    if err := m.GenericAPIServer.InstallLegacyAPIGroup(genericapiserver.DefaultLegacyAPIPrefix, &apiGroupInfo); err != nil {
        return fmt.Errorf("Error in registering group versions: %v", err)
    }
    return nil
}
```

m.GenericAPIServer.InstallLegacyAPIGroup 的调用链非常深，最终是为 Group 下每一个 API resources 注册 handler 及路由信息，其调用链为：m.GenericAPIServer.InstallLegacyAPIGroup --> s.installAPIResources --> apiGroupVersion.InstallREST --> installer.Install --> a.registerResourceHandlers 。其中几个方法的作用如下所示：

- s.installAPIResources ： 为每一个 API resource 调用 apiGroupVersion.InstallREST 添加路由；
- apiGroupVersion.InstallREST ： 将 restful.WebServic 对象添加到 container 中；
- installer.Install ： 返回最终的 restful.WebService 对象

a.registerResourceHandlers

该方法实现了 rest.Storage 到 restful.Route 的转换，其首先会判断 API Resource 所支持的 REST 接口，然后为 REST 接口添加对应的 handler，最后将其注册到路由中。

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/endpoints/installer.go:181

```
func (a *APIInstaller) registerResourceHandlers(path string, storage rest.Storage, ws *restful.WebService) (*metav1.APIResource, error) {
    admit := a.group.Admit

    .....

    // 1、判断该 resource 实现了哪些 REST 操作接口，以此来判断其支持的 verbs 以便为其添加路由
    creater, isCreater := storage.(rest.Creater)
    namedCreater, isNamedCreater := storage.(rest.NamedCreater)
    lister, isLister := storage.(rest.Lister)
    getter, isGetter := storage.(rest.Getter)
    getterWithOptions, isGetterWithOptions := storage.(rest.GetterWithOptions)
    gracefulDeleter, isGracefulDeleter := storage.(rest.GracefulDeleter)
    collectionDeleter, isCollectionDeleter := storage.(rest.CollectionDeleter)
    updater, isUpdater := storage.(rest.Updater)
    patcher, isPatcher := storage.(rest.Patcher)
    watcher, isWatcher := storage.(rest.Watcher)
    connecter, isConnecter := storage.(rest.Connecter)
    storageMeta, isMetadata := storage.(rest.StorageMetadata)
    storageVersionProvider, isStorageVersionProvider := storage.(rest.StorageVersionProvider)
    if !isMetadata {
        storageMeta = defaultStorageMetadata{}
    }
    exporter, isExporter := storage.(rest.Exporter)
    if !isExporter {
        exporter = nil
    }

```

```
28  ......
29
30  // 2、为 resource 添加对应的 actions 并根据是否支持 namespace
31  switch {
32  case !namespaceScoped:
33    ......
34
35    actions = appendIf(actions, action{"LIST", resourcePath, resourceParams, namer, false}, isLister)
36    actions = appendIf(actions, action{"POST", resourcePath, resourceParams, namer, false}, isCreater)
37    actions = appendIf(actions, action{"DELETECOLLECTION", resourcePath, resourceParams, namer, false}, isCollectionDeleter)
38    actions = appendIf(actions, action{"WATCHLIST", "watch/" + resourcePath, resourceParams, namer, false}, allowWatchList)
39
40    actions = appendIf(actions, action{"GET", itemPath, nameParams, namer, false}, isGetter)
41    if getSubpath {
42      actions = appendIf(actions, action{"GET", itemPath + "/{path:*}", proxyParams, namer, false}, isGetter)
43    }
44    actions = appendIf(actions, action{"PUT", itemPath, nameParams, namer, false}, isUpdater)
45    actions = appendIf(actions, action{"PATCH", itemPath, nameParams, namer, false}, isPatcher)
46    actions = appendIf(actions, action{"DELETE", itemPath, nameParams, namer, false}, isGracefulDeleter)
47    actions = appendIf(actions, action{"WATCH", "watch/" + itemPath, nameParams, namer, false}, isWatcher)
48    actions = appendIf(actions, action{"CONNECT", itemPath, nameParams, namer, false}, isConnecter)
49    actions = appendIf(actions, action{"CONNECT", itemPath + "/{path:*}", proxyParams, namer, false}, isConnecter && connectSubpath)
50  default:
51    ......
52    actions = appendIf(actions, action{"LIST", resourcePath, resourceParams, namer, false}, isLister)
53    actions = appendIf(actions, action{"POST", resourcePath, resourceParams, namer, false}, isCreater)
54    actions = appendIf(actions, action{"DELETECOLLECTION", resourcePath, resourceParams, namer, false}, isCollectionDeleter)
55    actions = appendIf(actions, action{"WATCHLIST", "watch/" + resourcePath, resourceParams, namer, false}, allowWatchList)
56
57    actions = appendIf(actions, action{"GET", itemPath, nameParams, namer, false}, isGetter)
58    ......
59  }
60
61  // 3、根据 action 创建对应的 route
62  kubeVerbs := map[string]struct{}{}
63  reqScope := handlers.RequestScope{
64    Serializer:     a.group.Serializer,
65    ParameterCodec:  a.group.ParameterCodec,
66    Creater:        a.group.Creater,
67    Convertor:      a.group.Convertor,
68    ......
69  }
70  ......
71  // 4、从 rest.Storage 到 restful.Route 映射
72  // 为每个操作添加对应的 handler
73  for _, action := range actions {
74    ......
75    verbOverrider, needOverride := storage.(StorageMetricsOverride)
76    switch action.Verb {
77    case "GET": ......
78    case "LIST":
79    case "PUT":
80    case "PATCH":
81    // 此处以 POST 操作进行说明
82    case "POST":
83      var handler restful.RouteFunction
84      // 5、初始化 handler
85      if isNamedCreater {
86        handler = restfulCreateNamedResource(namedCreater, reqScope, admit)
87      } else {
88        handler = restfulCreateResource(creater, reqScope, admit)
89      }
90      handler = metrics.InstrumentRouteFunc(action.Verb, group, version, resource, subresource, requestScope, metrics.APIServerCompc
91      article := GetArticleForNoun(kind, " ")
92      doc := "create" + article + kind
93      if isSubresource {
94        doc = "create " + subresource + " of" + article + kind
```

```
95              }
96              // 6、route 与 handler 进行绑定
97              route := ws.POST(action.Path).To(handler).
98                Doc(doc).
99                Param(ws.QueryParameter("pretty", "If 'true', then the output is pretty printed.")).
100               Operation("create"+namespaced+kind+strings.Title(subresource)+operationSuffix).
101               Produces(append(storageMeta.ProducesMIMETypes(action.Verb), mediaTypes...)...).
102               Returns(http.StatusOK, "OK", producedObject).
103               Returns(http.StatusCreated, "Created", producedObject).
104               Returns(http.StatusAccepted, "Accepted", producedObject).
105               Reads(defaultVersionedObject).
106               Writes(producedObject)
107             if err := AddObjectParams(ws, route, versionedCreateOptions); err != nil {
108               return nil, err
109             }
110             addParams(route, action.Params)
111             // 7、添加到路由中
112             routes = append(routes, route)
113           case "DELETE":
114           case "DELETECOLLECTION":
115           case "WATCH":
116           case "WATCHLIST":
117           case "CONNECT":
118           default:
119       }
120       ......
121       return &apiResource, nil
122   }
```

restfulCreateNamedResource

restfulCreateNamedResource 是 POST 操作对应的 handler，最终会调用 createHandler 方法完成。

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/endpoints/installer.go:1087

```
1   func restfulCreateNamedResource(r rest.NamedCreater, scope handlers.RequestScope, admit admission.Interface) restful.RouteFunction {
2     return func(req *restful.Request, res *restful.Response) {
3       handlers.CreateNamedResource(r, &scope, admit)(res.ResponseWriter, req.Request)
4     }
5   }
6
7   func CreateNamedResource(r rest.NamedCreater, scope *RequestScope, admission admission.Interface) http.HandlerFunc {
8     return createHandler(r, scope, admission, true)
9   }
```

createHandler

createHandler 是将数据写入到后端存储的方法，对于资源的操作都有相关的权限控制，在 createHandler 中首先会执行 decoder 和 admission 操作，然后调用 create 方法完成 resource 的创建，在 create 方法中会进行 validate 以及最终将数据保存到后端存储中。 admit 操作即执行 kube-apiserver 中的 admission-plugins，admission-plugins 在 CreateKubeAPIServerConfig 中被初始化为了 admissionChain，其初始化的调用链为 CreateKubeAPIServerConfig --> buildGenericConfig --> s.Admission.ApplyTo --> a.GenericAdmission.ApplyTo --> a.Plugins.NewFromPlugins ，最终在 a.Plugins.NewFromPlugins 中将所有已启用的 plugins 封装为 admissionChain，此处要执行的 admit 操作即执行 admission-plugins 中的 admit 操作。

createHandler 中调用的 create 方法是 genericregistry.Store 对象的方法，在每个 resource 初始化 RESTStorage 都会引入 genericregistry.Store 对象。

createHandler 中所有的操作就是本文开头提到的请求流程，如下所示：

```
1   v1beta1 ⇒ internal ⇒  |  ⇒    |  ⇒ v1 ⇒ json/yaml ⇒ etcd
2                admission   validation
```

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/endpoints/handlers/create.go:46

```
func createHandler(r rest.NamedCreater, scope *RequestScope, admit admission.Interface, includeName bool) http.HandlerFunc {
  return func(w http.ResponseWriter, req *http.Request) {
    trace := utiltrace.New("Create", utiltrace.Field{"url", req.URL.Path})
    defer trace.LogIfLong(500 * time.Millisecond)
    ......

    gv := scope.Kind.GroupVersion()

    // 1、得到合适的SerializerInfo
    s, err := negotiation.NegotiateInputSerializer(req, false, scope.Serializer)
    if err != nil {
      scope.err(err, w, req)
      return
    }
    // 2、找到合适的 decoder
    decoder := scope.Serializer.DecoderToVersion(s.Serializer, scope.HubGroupVersion)

    body, err := limitedReadBody(req, scope.MaxRequestBodyBytes)
    if err != nil {
      scope.err(err, w, req)
      return
    }

    ......

    defaultGVK := scope.Kind
    original := r.New()
    trace.Step("About to convert to expected version")
    // 3、decoder 解码
    obj, gvk, err := decoder.Decode(body, &defaultGVK, original)
    ......

    ae := request.AuditEventFrom(ctx)
    admit = admission.WithAudit(admit, ae)
    audit.LogRequestObject(ae, obj, scope.Resource, scope.Subresource, scope.Serializer)

    userInfo, _ := request.UserFrom(ctx)


    if len(name) == 0 {
      _, name, _ = scope.Namer.ObjectName(obj)
    }
    // 4、执行 admit 操作，即执行 kube-apiserver 启动时加载的 admission-plugins，
    admissionAttributes := admission.NewAttributesRecord(......)
    if mutatingAdmission, ok := admit.(admission.MutationInterface); ok && mutatingAdmission.Handles(admission.Create) {
      err = mutatingAdmission.Admit(ctx, admissionAttributes, scope)
      if err != nil {
        scope.err(err, w, req)
        return
      }
    }

    ......
    // 5、执行 create 操作
    result, err := finishRequest(timeout, func() (runtime.Object, error) {
      return r.Create(
        ctx,
        name,
        obj,
        rest.AdmissionToValidateObjectFunc(admit, admissionAttributes, scope),
        options,
      )
    })
    ......
  }
```

```
64      }
65  }
```

总结

本文主要分析 kube-apiserver 的启动流程，kube-apiserver 中包含三个 server，分别为 KubeAPIServer、APIExtensionsServer 以及 AggregatorServer，三个 server 是通过委托模式连接在一起的，初始化过程都是类似的，首先为每个 server 创建对应的 config，然后初始化 http server，http server 的初始化过程为首先初始化 GoRestfulContainer ，然后安装 server 所包含的 API，安装 API 时首先为每个 API Resource 创建对应的后端存储 RESTStorage，再为每个 API Resource 支持的 verbs 添加对应的 handler，并将 handler 注册到 route 中，最后将 route 注册到 webservice 中，启动流程中 RESTFul API 的实现流程是其核心，至于 kube-apiserver 中认证鉴权等 filter 的实现、多版本资源转换、kubernetes service 的实现等一些细节会在后面的文章中继续进行分析。

参考：

https://mp.weixin.qq.com/s/hTEWatYLhTnC5X0FBM2RWQ

https://bbbmj.github.io/2019/04/13/Kubernetes/code-analytics/kube-apiserver/

https://mp.weixin.qq.com/s/TQuqAAzBjeWHwKPJZ3iJhA

https://blog.openshift.com/kubernetes-deep-dive-api-server-part-1/

https://www.jianshu.com/p/daa4ff387a78

kube-apiserver

‹ kubelet 中垃圾回收机制的设计与实现

kube-apiserver 中 apiserver service 的实现 ›

1 评论

辉辉乎 Chrome 80.0.3987.122 Windows 10.0

2020-03-06 回复

真是高质量的系列文章，受教了！

谢谢。

- 3、 autoRegistrationController： 用于保持 API 中存在的一组特定的 APIServices；
- 4、 crdRegistrationController： 负责将 CRD GroupVersions 自动注册到 APIServices 中；
- 5、 openAPIAggregationController： 将 APIServices 资源的变化同步至提供的 OpenAPI 文档；

kubernetes 中的一些附加组件，比如 metrics-server 就是通过 Aggregator 的方式进行扩展的，实际环境中可以通过使用 apiserver-builder 工具轻松以 Aggregator 的扩展方式创建自定义资源。

启用 API Aggregation

在 kube-apiserver 中需要增加以下配置来开启 API Aggregation：

```
--proxy-client-cert-file=/etc/kubernetes/certs/proxy.crt
--proxy-client-key-file=/etc/kubernetes/certs/proxy.key
--requestheader-client-ca-file=/etc/kubernetes/certs/proxy-ca.crt
--requestheader-allowed-names=aggregator
--requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group
--requestheader-username-headers=X-Remote-User
```

KubeAPIServer

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

APIExtensionServer

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

其中包含的 controller 以及功能如下所示：

- 1、 openapiController： 将 crd 资源的变化同步至提供的 OpenAPI 文档，可通过访问 `/openapi/v2` 进行查看；
- 2、 crdController： 负责将 crd 信息注册到 apiVersions 和 apiResources 中，两者的信息可通过 `$ kubectl api-versions` 和 `$ kubectl api-resources` 查看；

- 3、调用 createAPIExtensionsServer 创建 apiExtensionsServer 实例；
- 4、调用 CreateKubeAPIServer 初始化 kubeAPIServer；
- 5、调用 createAggregatorConfig 为 aggregatorServer 创建配置并调用 createAggregatorServer 初始化 aggregatorServer；
- 6、配置并判断是否启动非安全的 http server；

k8s.io/kubernetes/cmd/kube-apiserver/app/server.go:165

```
1  func CreateServerChain(completedOptions completedServerRunOptions, stopCh <-chan struct{}) (*aggregatorapiserver.APIAggregator, errc
2    nodeTunneler, proxyTransport, err := CreateNodeDialer(completedOptions)
3    if err != nil {
4      return nil, err
5    }
6    // 1、为 kubeAPIServer 创建配置
7    kubeAPIServerConfig, insecureServingInfo, serviceResolver, pluginInitializer, admissionPostStartHook, err :=                CreateKu
8    if err != nil {
9      return nil, err
10   }
11
12   // 2、判断是否配置了 APIExtensionsServer，创建 apiExtensionsConfig
13   apiExtensionsConfig, err := createAPIExtensionsConfig(*kubeAPIServerConfig.GenericConfig, kubeAPIServerConfig.ExtraConfig.Versioned
14     serviceResolver, webhook.NewDefaultAuthenticationInfoResolverWrapper(proxyTransport, kubeAPIServerConfig.GenericConfig.Loopb
15   if err != nil {
16     return nil, err
17   }
18
19   // 3、初始化 APIExtensionsServer
20   apiExtensionsServer, err := createAPIExtensionsServer(apiExtensionsConfig, genericapiserver.NewEmptyDelegate())
21   if err != nil {
22     return nil, err
23   }
24
25   // 4、初始化 KubeAPIServer
26   kubeAPIServer, err := CreateKubeAPIServer(kubeAPIServerConfig, apiExtensionsServer.GenericAPIServer, admissionPostStartHook)
27   if err != nil {
28     return nil, err
29   }
30
31   // 5、创建 AggregatorConfig
32   aggregatorConfig, err := createAggregatorConfig(*kubeAPIServerConfig.GenericConfig, completedOptions.ServerRunOptions, kubeAPISe
33   if err != nil {
34     return nil, err
35   }
36
37   // 6、初始化 AggregatorServer
38   aggregatorServer, err := createAggregatorServer(aggregatorConfig, kubeAPIServer.GenericAPIServer, apiExtensionsServer.Informers)
39   if err != nil {
40     return nil, err
41   }
42
43   // 7、判断是否启动非安全端口的 http server
44   if insecureServingInfo != nil {
```

- 3、初始化 server 中需要使用的 controller，主要有 openapiController 、 crdController 、 namingController 、 establishingController 、 nonStructuralSchemaController 、 apiApprovalController 、 finalizingControlle r;
- 4、将需要启动的 controller 以及 informer 添加到 PostStartHook 中；

k8s.io/kubernetes/cmd/kube-apiserver/app/apiextensions.go:94

```
func createAPIExtensionsServer(apiextensionsConfig *apiextensionsapiserver.Config, delegateAPIServer genericapiserver.DelegationTarget)
  return apiextensionsConfig.Complete().New(delegateAPIServer)
}
```

k8s.io/kubernetes/staging/src/k8s.io/apiextensions-apiserver/pkg/apiserver/apiserver.go:132

```
func (c completedConfig) New(delegationTarget genericapiserver.DelegationTarget) (*CustomResourceDefinitions, error) {
  // 1、初始化 genericServer
  genericServer, err := c.GenericConfig.New("apiextensions-apiserver", delegationTarget)
  if err != nil {
    return nil, err
  }

  s := &CustomResourceDefinitions{
    GenericAPIServer: genericServer,
  }

  // 2、初始化 APIGroup Info，APIGroup 指该 server 需要暴露的 API
  apiResourceConfig := c.GenericConfig.MergedResourceConfig
  apiGroupInfo := genericapiserver.NewDefaultAPIGroupInfo(apiextensions.GroupName, Scheme, metav1.ParameterCodec, Codecs)
  if apiResourceConfig.VersionEnabled(v1beta1.SchemeGroupVersion) {
    storage := map[string]rest.Storage{}
    customResourceDefintionStorage := customresourcedefinition.NewREST(Scheme, c.GenericConfig.RESTOptionsGetter)
    storage["customresourcedefinitions"] = customResourceDefintionStorage
    storage["customresourcedefinitions/status"] = customresourcedefinition.NewStatusREST(Scheme, customResourceDefintionStorage)

    apiGroupInfo.VersionedResourcesStorageMap[v1beta1.SchemeGroupVersion.Version] = storage
  }
  if apiResourceConfig.VersionEnabled(v1.SchemeGroupVersion) {
    ......
  }

  // 3、注册 APIGroup
  if err := s.GenericAPIServer.InstallAPIGroup(&apiGroupInfo); err != nil {
    return nil, err
  }

  // 4、初始化需要使用的 controller
  crdClient, err := internalclientset.NewForConfig(s.GenericAPIServer.LoopbackClientConfig)
  if err != nil {
    return nil, fmt.Errorf("failed to create clientset: %v", err)
  }
  s.Informers = internalinformers.NewSharedInformerFactory(crdClient, 5*time.Minute)
```

k8s.io/kubernetes/staging/src/k8s.io/apiserver/pkg/registry/generic/registry/store.go:1192

```
func (e *Store) CompleteWithOptions(options *generic.StoreOptions) error {
    ......

    var isNamespaced bool
    switch {
    case e.CreateStrategy != nil:
        isNamespaced = e.CreateStrategy.NamespaceScoped()
    case e.UpdateStrategy != nil:
        isNamespaced = e.UpdateStrategy.NamespaceScoped()
    default:
        return fmt.Errorf("store for %s must have CreateStrategy or UpdateStrategy set", e.DefaultQualifiedResource.String())
    }
    ......

    // 1、调用 options.RESTOptions.GetRESTOptions
    opts, err := options.RESTOptions.GetRESTOptions(e.DefaultQualifiedResource)
    if err != nil {
        return err
    }

    // 2、设置 ResourcePrefix
    prefix := opts.ResourcePrefix
    if !strings.HasPrefix(prefix, "/") {
        prefix = "/" + prefix
    }

    if prefix == "/" {
        return fmt.Errorf("store for %s has an invalid prefix %q", e.DefaultQualifiedResource.String(), opts.ResourcePrefix)
    }

    if e.KeyRootFunc == nil && e.KeyFunc == nil {
        ......
    }

    keyFunc := func(obj runtime.Object) (string, error) {
        ......
    }

    // 3、以下操作主要是将 opts 对象中的值赋值到 store 对象中
    if e.DeleteCollectionWorkers == 0 {
        e.DeleteCollectionWorkers = opts.DeleteCollectionWorkers
    }

    e.EnableGarbageCollection = opts.EnableGarbageCollection
    if e.ObjectNameFunc == nil {
        ......

    }

    if e.Storage.Storage == nil {
```