极客时间 训练营

模块六：Client-go 入门和实战

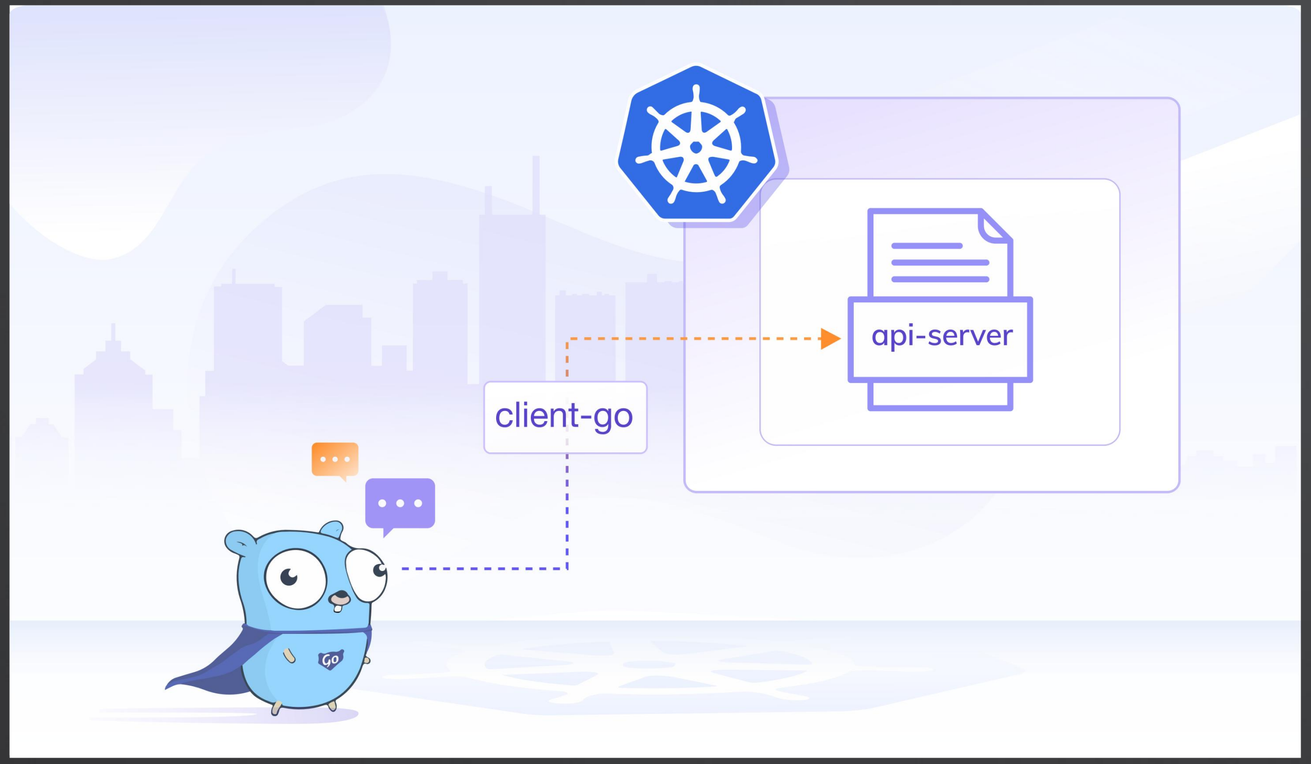
王炜/ 前腾讯云 CODING 高级架构师

目录

1. Client-go 简介

什么是 Client-go?

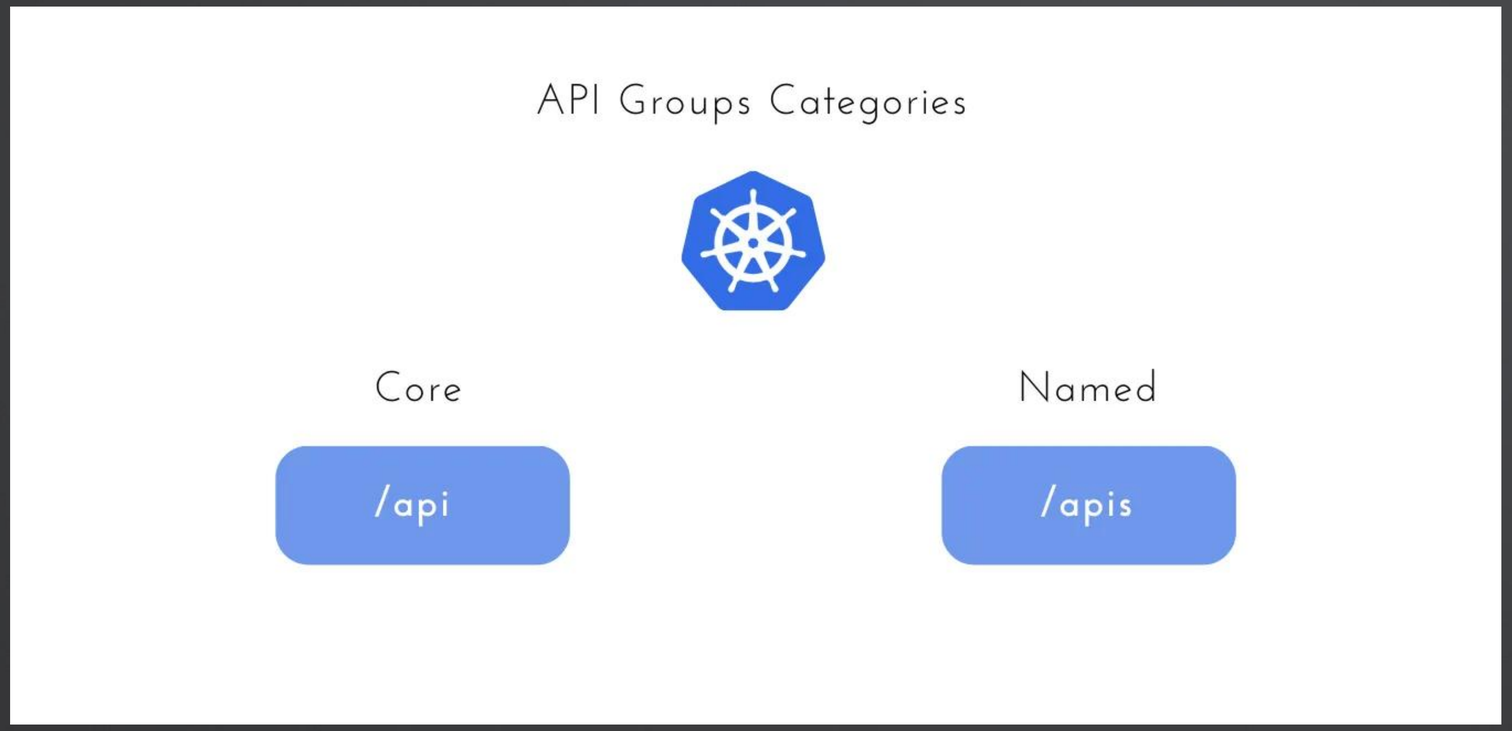
Client-go 是 Kubernetes 官方提供的 Go 语言的客户端库， 使用该库可以访问 Kubernetes API Server，实现对 Kubernetes 资源（包括 CRD）的增删改查操作。



Kubernetes API Group

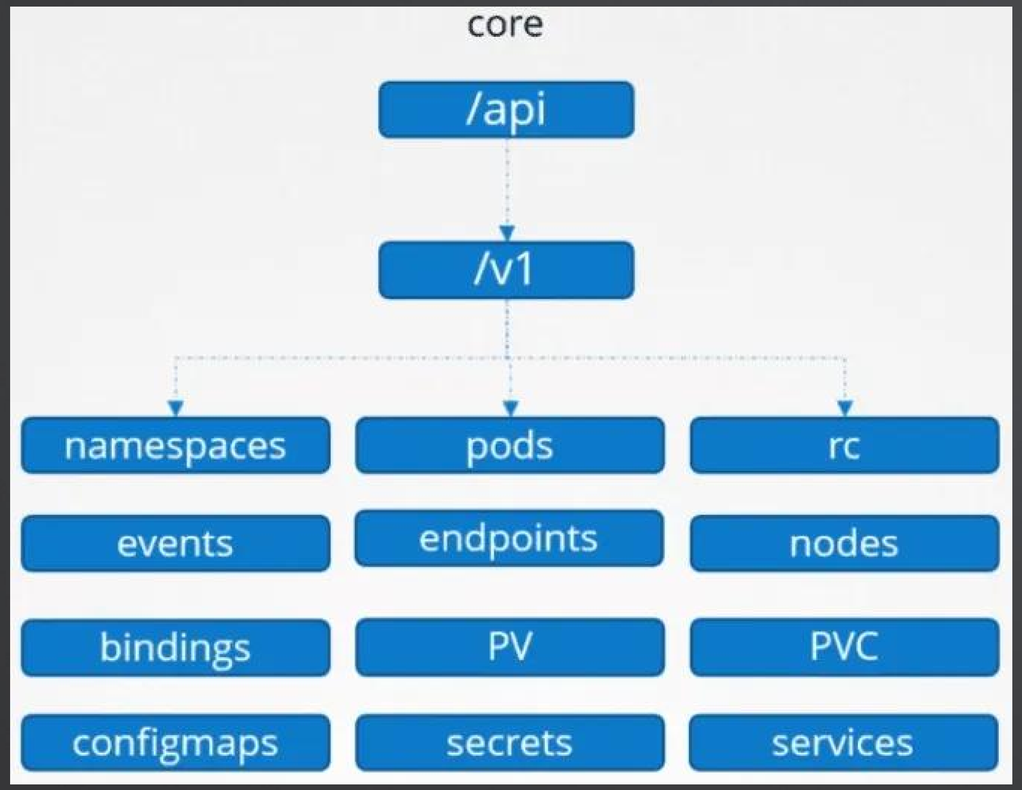
K8s API 分为核心 API 和命名 API 组，对资源的增删改查可以理解为是对 K8s API 的请求

Client-go 封装了请求 API 所需的鉴权、请求参数以及将返回 JSON 转化为 Go 结构体（Unmarshaling）



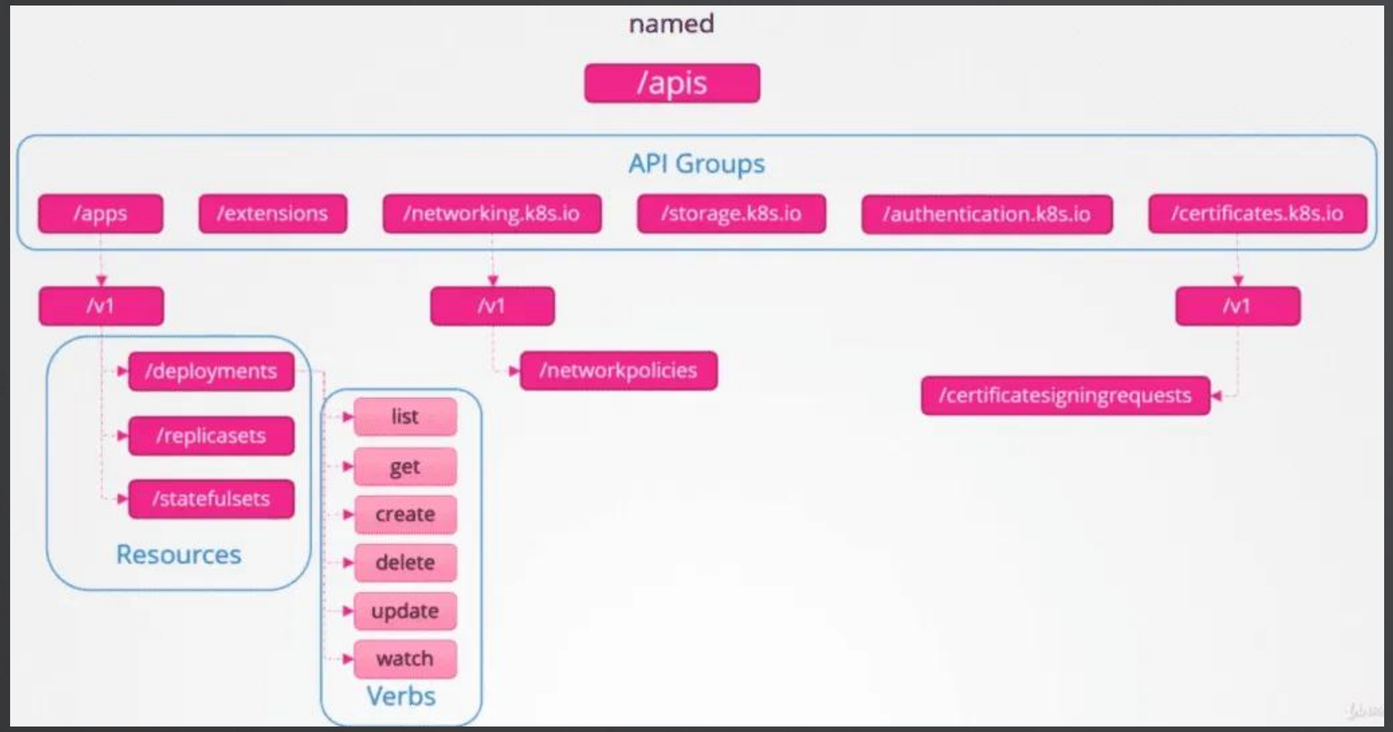
Kubernetes Core API

包括 namespace、 service、pods、nodes、configmap、secrets、pv 等对象



Kubernetes Named API

包括 apps/v1/deployments、 apps/v1/replicasets 等对象



验证 K8s API 请求

获取 namespaces(core API)：https://127.0.0.1:62306/api/v1/namespaces?limit=500

ns.sh

```
> kubectl get ns -v6
I0829 18:07:24.228179   97471 round_trippers.go:553] GET
https://127.0.0.1:62306/api/v1/namespaces?limit=500 200 OK in 18 milliseconds
NAME                 STATUS   AGE
default              Active   66m
kube-node-lease      Active   66m
kube-public          Active   66m
kube-system          Active   66m
local-path-storage   Active   65m
```

获取 deployment(named API)：https://127.0.0.1:62306/apis/apps/v1/namespaces/default/deployments?limit=500

deployments.sh

```
> kubectl get deployment -v6
I0829 18:18:12.772128   97763 round_trippers.go:553] GET
https://127.0.0.1:62306/apis/apps/v1/namespaces/default/deployments?limit=500
200 OK in 25 milliseconds
No resources found in default namespace.
```

理解 GVK 和 GVR

- GVK：Group、Version、Kind
  - {Group: "apps", Version: "v1", Kind: "Deployment"}
- GVR：Group、Version、Resource
  - {Group: "apps", Version: "v1", Resource: "deployments"}
  - https://127.0.0.1:62306/apis/apps/v1/namespaces/default/deployments?limit=500

deployments.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
```

deployments.sh

```
https://127.0.0.1:62306/apis/apps/v1/namespaces/default/deployments
```

每个资源的 GVK 都可以转成对应的 Rest API 请求（GVR）

Quick Start

以获取 Pod 和 Deployment 为例介绍 Client-go 的用法

deployments.go

```go
kubeconfig := flag.String("kubeconfig", "/Users/wangwei/.kube/config", "location of kubeconfig file")

config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)

clientset, err := kubernetes.NewForConfig(config)

pods, err := clientset.CoreV1().Pods("default").List(context.Background(), metav1.ListOptions{})

for _, pod := range pods.Items {
  fmt.Printf("Pod name %s\n", pod.Name)
}
```

2. Client-go In Cluster Configuration

In Cluster Configuration

- 借助 InClusterConfig 方法来获取请求 K8s API 的 Token
- 需要注意给 Service Account 授权，否则无对应权限

incluster.go

```
1    config, err := rest.InClusterConfig()
2    if err != nil {
3      fmt.Printf("error %s", err.Error())
4    }
5
6    clientset, err := kubernetes.NewForConfig(config)
7    if err != nil {
8      fmt.Printf("error %s", err.Error())
9    }
```

In Cluster Configuration 原理

- K8s 会给每个 Pod 注入 Service Account 配置文件，包括 token 和 ca 证书
  - /var/run/secrets/kubernetes.io/serviceaccount/token
  - /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
- K8s 默认会给所有 Pod 注入 KUBERNETES_SERVICE_HOST 和 KUBERNETES_SERVICE_PORT 变量

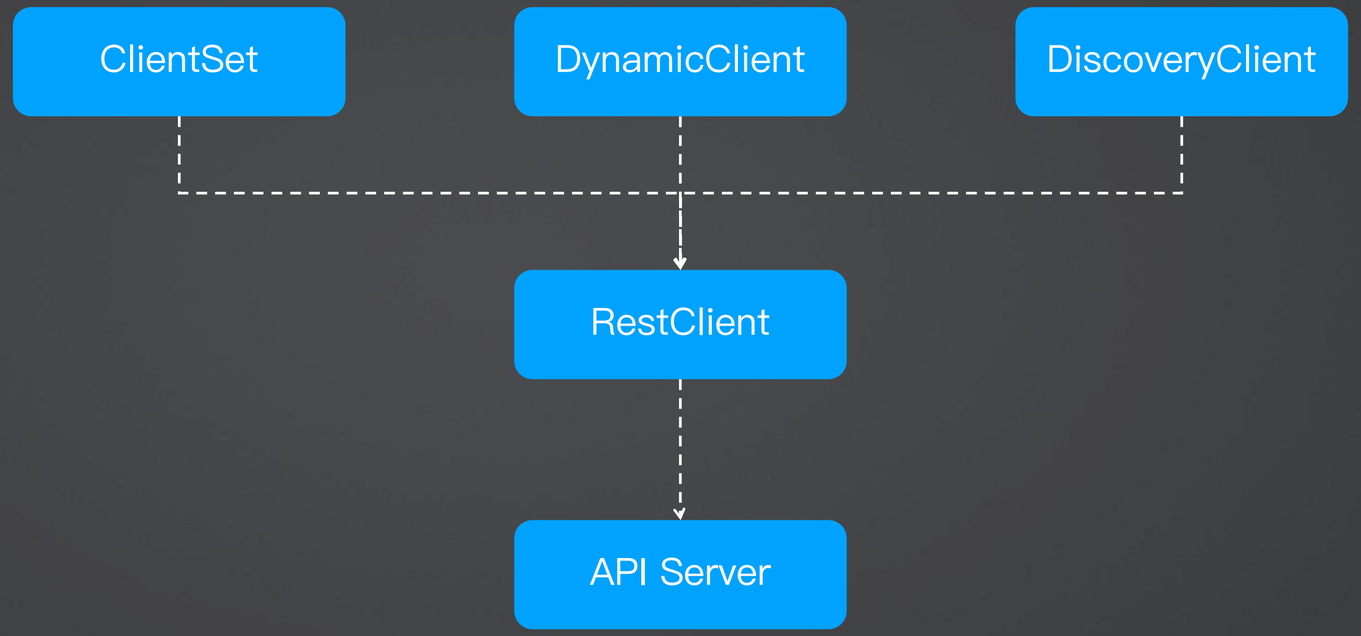
```
NJS_RELEASE=1~bookworm
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
#
```

```
# pwd
/var/run/secrets/kubernetes.io/serviceaccount
# ls
ca.crt  namespace  token
```

3. ClientSet、DynamicClient、RESTClient、DiscoveryClient

Client–go 四种 Client

1. RestClient
2. ClientSet
3. DynamicClient
4. DiscoveryClient



RestClient

- 源码：https://github.com/kubernetes/client-go/blob/master/rest/client.go
- 最底层的客户端，直接跟 Rest API 交互

incluster.go

```go
type Interface interface {
    GetRateLimiter() flowcontrol.RateLimiter
    Verb(verb string) *Request
    Post() *Request
    Put() *Request
    Patch(pt types.PatchType) *Request
    Get() *Request
    Delete() *Request
    APIVersion() schema.GroupVersion
}
```

ClientSet

- 源码：https://github.com/kubernetes/client-go/blob/master/kubernetes/clientset.go
- 最常用的 Client，实现了所有 K8s 标准对象的接口

clientset.go

```
type Interface interface {
  Discovery() discovery.DiscoveryInterface
  AdmissionregistrationV1()
admissionregistrationv1.AdmissionregistrationV1Interface
  AdmissionregistrationV1alpha1()
admissionregistrationv1alpha1.AdmissionregistrationV1alpha1Interface
  AdmissionregistrationV1beta1()
admissionregistrationv1beta1.AdmissionregistrationV1beta1Interface
  InternalV1alpha1() internalv1alpha1.InternalV1alpha1Interface
  AppsV1() appsv1.AppsV1Interface
  AppsV1beta1() appsv1beta1.AppsV1beta1Interface
  AppsV1beta2() appsv1beta2.AppsV1beta2Interface
  ......
}
```

DynamicClient

- 源码：https://github.com/kubernetes/client-go/tree/master/dynamic
- 动态客户端，可以对任何资源进行操作（包括 CRD）

dynamic.go

```go
type ResourceInterface interface {
  Create(ctx context.Context, obj *unstructured.Unstructured, options metav1.CreateOptions, subresources ...string) (*unstructured.Unstructured, error)
  Update(ctx context.Context, obj *unstructured.Unstructured, options metav1.UpdateOptions, subresources ...string) (*unstructured.Unstructured, error)
  UpdateStatus(ctx context.Context, obj *unstructured.Unstructured, options metav1.UpdateOptions) (*unstructured.Unstructured, error)
  Delete(ctx context.Context, name string, options metav1.DeleteOptions, subresources ...string) error
  DeleteCollection(ctx context.Context, options metav1.DeleteOptions, listOptions metav1.ListOptions) error
  Get(ctx context.Context, name string, options metav1.GetOptions, subresources ...string) (*unstructured.Unstructured, error)
  List(ctx context.Context, opts metav1.ListOptions) (*unstructured.UnstructuredList, error)
  Watch(ctx context.Context, opts metav1.ListOptions) (watch.Interface, error)
  Patch(ctx context.Context, name string, pt types.PatchType, data []byte, options metav1.PatchOptions, subresources ...string) (*unstructured.Unstructured, error)
  Apply(ctx context.Context, name string, obj *unstructured.Unstructured, options metav1.ApplyOptions, subresources ...string) (*unstructured.Unstructured, error)
  ApplyStatus(ctx context.Context, name string, obj *unstructured.Unstructured, options metav1.ApplyOptions) (*unstructured.Unstructured, error)
}
```

dynamic.go

```go
type Unstructured struct {
  // Object is a JSON compatible map with string, float, int, bool, []interface{}, or
  // map[string]interface{}
  // children.
  Object map[string]interface{}
}
```

DynamicClient

- 可通过 GVR 来创建任何资源

dynamic.go

```go
deployObj := &unstructured.Unstructured{}
  if err := yaml.Unmarshal([]byte(deployYaml), deployObj); err != nil {
    panic(err)
  }

  // 从deployObj中提取apiVersion和kind以确定GVR
  apiVersion, found, err := unstructured.NestedString(deployObj.Object,
"apiVersion")
  if err != nil || !found {
    log.Fatalln("apiVersion not found:", err)
  }

  kind, found, err := unstructured.NestedString(deployObj.Object, "kind")
  if err != nil || !found {
    log.Fatalln("kind not found:", err)
  }
```

DiscoveryClient

- DiscoveryClient：发现客户端，主要用于发现 apiserver 支持的 Group、Version、Resource
- kubectl 的 api-version 和 api-resource 就是通过 DiscoveryClient 实现的，它可以将信息缓存在本地 Cache，以减轻 API 的访问压力，默认在 ~/.kube/cache/discovery 目录

```
> ls discovery
101.32.40.234_6443   127.0.0.1_56879      43.129.168.91_6443   43.134.215.144_6443
119.28.202.101_6443  127.0.0.1_62156      43.129.180.37_6443   43.135.18.17_6443
124.156.160.159_6443 127.0.0.1_62306      43.129.188.152_6443  43.154.132.104_6443
127.0.0.1_50441      127.0.0.1_62499      43.129.189.227_6443  43.154.206.12_6443
127.0.0.1_51797      129.226.225.166_6443 43.129.205.233_6443  43.155.17.178_6443
127.0.0.1_54794      43.128.18.220_6443   43.129.207.132_6443
> cd 6125de0a33d936c7d4dc45067fe99924
> ls
admissionregistration.k8s.io coordination.k8s.io          node.k8s.io
apiextensions.k8s.io         discovery.k8s.io             policy
apiregistration.k8s.io       events.k8s.io                rbac.authorization.k8s.io
apps                         flowcontrol.apiserver.k8s.io scheduling.k8s.io
authentication.k8s.io        helm.cattle.io               servergroups.json
authorization.k8s.io         k3s.cattle.io                storage.k8s.io
autoscaling                  metrics.k8s.io               v1
batch                        monitoring.coreos.com
certificates.k8s.io          networking.k8s.io
```

4. 实现一个 Client-go Watch 客户端

概述

watch.go

```go
watcher, _ := clientset.CoreV1().Namespaces().Watch(context.Background(),
metav1.ListOptions{TimeoutSeconds: &timeOut})
for event := range watcher.ResultChan() {
    item := event.Object.(*corev1.Namespace)

    switch event.Type {
    case watch.Modified:
    case watch.Bookmark:
    case watch.Error:
    case watch.Deleted:
    case watch.Added:
        processNamespace(item.GetName())
    }
}

```

- Watch 指的是持续监听特定的资源变化，包括增删改查
- 使用 ClientSet 提供的 Watch 方法监听事件
- 一旦产生事件，则可以执行对应的业务逻辑

为什么不推荐直接使用 Watch

- 处理 Watch 超时、断开重连等情况的处理比较复杂
- Watch 机制直接请求 K8s API Server，增加了集群负载
- 对于希望对多个资源 Watch 时需创建单独的连接而无法共享，增加资源消耗和集群负载
- 重连可能会导致事件丢失
- 大量事件产生时无限流逻辑，可能导致业务过载崩溃
- 业务获得事件信号后，如果处理失败，没有第二次处理机会

使用 Informer 代替

- 基于 Watch 实现，提供更高层次的抽象，更简单、安全、高性能
- 自动处理超时和重连机制
- 本地缓存机制，无需频繁调用 API Server
- 内置全量和增量同步机制，确保事件不丢失
- 可结合 Rate Limiting 和延迟队列，控制事件处理速率，避免业务过载，同时支持错误重试

使用 Informer 代替

- 使用 SharedInformerFactory 创建一个共享的 Informer 实例
- 减少网络和资源消耗，减轻 K8s API 负载

informer.go

```go
// 初始化 informer
informerFactory := informers.NewSharedInformerFactory(clientset, time.Hour*12)

// 对 Deployment 监听
deployInformer := informerFactory.Apps().V1().Deployments()
informer := deployInformer.Informer()
informer.AddEventHandler(cache.ResourceEventHandlerFuncs{
  AddFunc:    onAddDeployment,
  UpdateFunc: onUpdateDeployment,
  DeleteFunc: onDeleteDeployment,
})

// 对 Service 监听
serviceInformer := informerFactory.Core().V1().Services()
serviceInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
  AddFunc:    onAddService,
  UpdateFunc: onUpdateService,
  DeleteFunc: onDeleteService,
})
```

引入 RateLimitingQueue

- Q：上一个例子的 EventHandler 业务逻辑处理失败怎么办?
- A：基于事件驱动，没有二次处理机会
- 引入 WorkQueue(RateLimitingQueue) 处理业务逻辑：错误重试、防止 Hot Loop（过载）

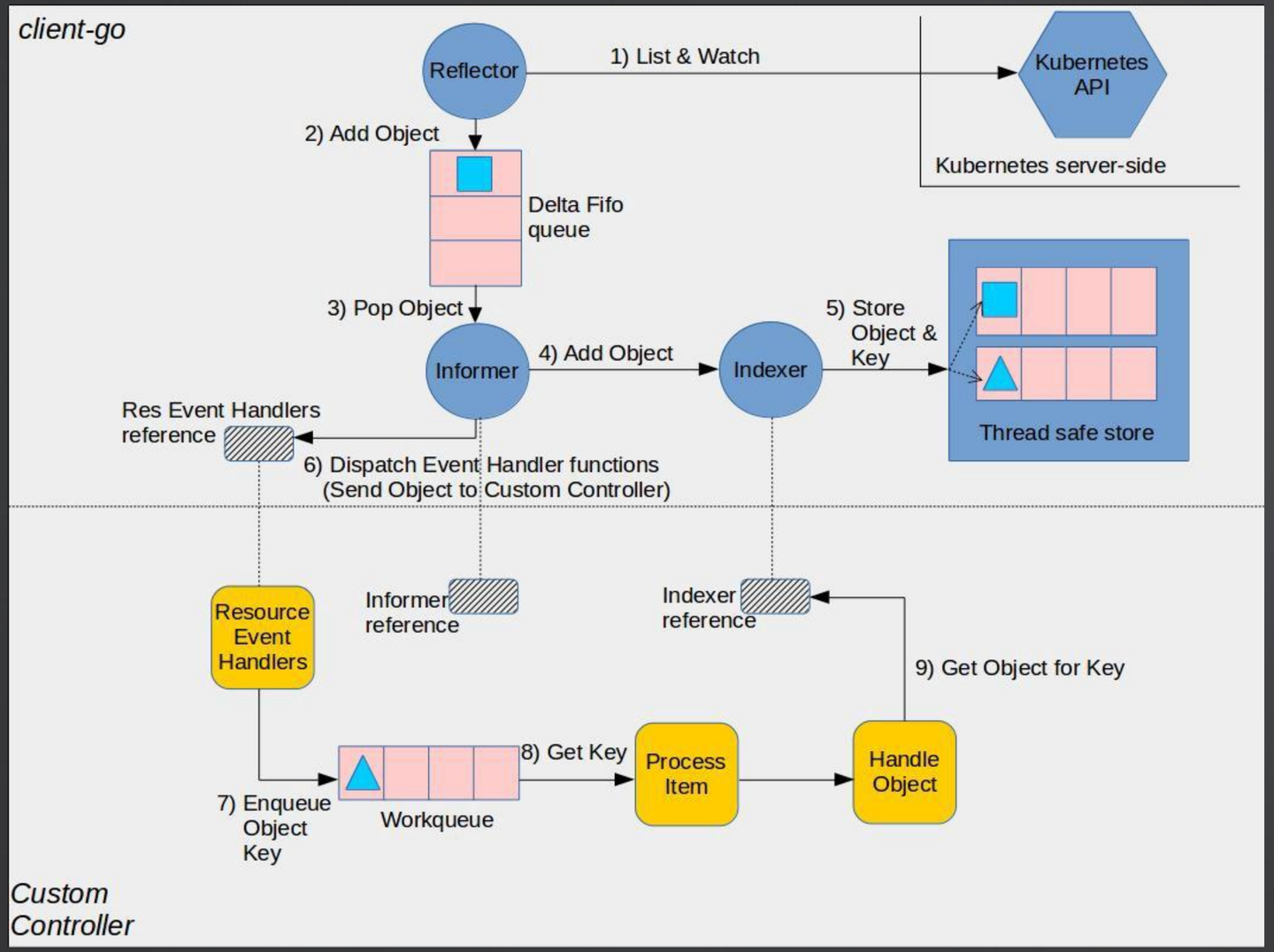
queue.go

```go
// 创建速率限制队列
queue := workqueue.NewTypedRateLimitingQueue(workqueue.DefaultTypedControllerRateLimiter[string]())

// 对 Deployment 监听
deployInformer := informerFactory.Apps().V1().Deployments()
informer := deployInformer.Informer()
informer.AddEventHandler(cache.ResourceEventHandlerFuncs{
  AddFunc:    func(obj interface{}) { onAddDeployment(obj, queue) },
  UpdateFunc: func(old, new interface{}) { onUpdateDeployment(new, queue) },
  DeleteFunc: func(obj interface{}) { onDeleteDeployment(obj, queue) },
})
```

5. 进阶：Informers、Indexer、WorkQueue

Informer 架构



- Reflector：借助 List/Watch 机制监听资源变化，并将对象放到 Delta Fifo 队列里
- Informer：从 Delta Fifo 队列里弹出对象，然后将对象缓存到 Indexer 里（线程安全的 Map）
- Indexer：提供对象的检索能力，比如可以通过 indexer.GetByKey(key) 来获取缓存中的任何一个对象（通过 cache.MetaNamespaceKeyFunc 生成 key: <namespace>/<name>）

Reflector DeltaFifo

Reflector.go

```go
type DeltaType string

const (
 Added   DeltaType = "Added"
 Updated DeltaType = "Updated"
 Deleted DeltaType = "Deleted"
 Replaced DeltaType = "Replaced"
 Sync DeltaType = "Sync"
)

type Delta struct {
 Type   DeltaType
 Object interface{}
}
```

- Delta 结构体包括：
  - 操作类型：增删改同步
  - Interface{} 对象
- Fifo：先进先出队列

Indexer

- Indexers：存储索引器，key 为索引器名称，value 为索引器实现的函数
- IndexFunc：计算 obj 用于索引的 key，client-go 默认是 MetaNamespaceIndexFunc（之前例子用到的）
- Indices：存储 Index 类型名和对应类型的 Index 的映射
- Index：存储缓存数据，其结构为 K/V

indexer.go

```go
// Indexers maps a name to a IndexFunc
type Indexers map[string]IndexFunc
// IndexFunc knows how to compute the set of indexed values for an object.
type IndexFunc func(obj interface{}) ([]string, error)
// Indices maps a name to an Index
type Indices map[string]Index
// Index maps the indexed value to a set of keys in the store that match on that value
type Index map[string]sets.String
```

WorkQueue

- Interface：通用先进先出队列，支持去重机制
- DelayingInterface：延迟队列接口，基于 Interface 接口封装，延迟一段时间后再将元素存入队列
- RateLimitingInterface：限速队列接口，基于 DelayingInterface 接口封装，比较常用
  - 失败重试，指数退避策略
  - 限速，避免业务被打爆

workqueue.go

```go
// k8s.io/client-go/util/workqueue/queue.go
type Interface interface {
  Add(item interface{})
  Len() int
  Get() (item interface{}, shutdown bool)
  Done(item interface{})
  ShutDown()
  ShuttingDown() bool
}
```

6. 实现一个简单的 Kubectl get CRD

kubectl get CRD

- CRD 无法通过 ClientSet 获取，需通过 dynamicClient 获取
- get crd_name (GVK) 转化为 K8s API 请求 (GVR) 的过程
- 借助 RESTMapping 进行转化

gvk.go

```go
gvk := schema.GroupVersionKind{
  Group:   "mygroup.example.com",
  Version: "v1alpha1",
  Kind:    kind,
}

mapping, err := mapper.RESTMapping(gvk.GroupKind(), gvk.Version)
if err != nil {
  panic(err)
}
// mapping.Resource 就是 GVR，这样就实现 GVK->GVR 的转化
```

Example CRD

crd.yaml

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: myresources.mygroup.example.com
spec:
  group: mygroup.example.com
  versions:
    - name: v1alpha1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                field1:
                  type: string
                  description: First example field
                field2:
                  type: string
                  description: Second example field
            status:
              type: object
  scope: Namespaced
  names:
    plural: myresources
    singular: myresource
    kind: MyResource
    shortNames:
      - myres
```

resource.yaml

```yaml
apiVersion: mygroup.example.com/v1alpha1
kind: MyResource
metadata:
  name: my-resource-instance
  namespace: default
spec:
  field1: "ExampleValue1"
  field2: "ExampleValue2"
```

RestMapping 原理

- NewDiscoveryRESTMapper: 用 CRD 定义的 plural(复数) 和 singular(单数) 字段，建立 Kind 和 Resource 的关系
- 通过 RestMapper.RESTMapping 方法来实现 GVK 和 GVR 的转化

restmapping.go

```
1  mapper := restmapper.NewDiscoveryRESTMapper(apiGroupResources)
2
3  // NewDiscoveryRESTMapper 实现
4  func NewDiscoveryRESTMapper(groupResources []*APIGroupResources)
   meta.RESTMapper{
5    ......
6    // 获取 CRD 的 plural 和 singular 字段，用来建立 Kind 和 Resource 的关系
7    plural := gv.WithResource(resource.Name)
8    singular := gv.WithResource(resource.SingularName)
9  }
```

THANKS