




# Clientset 使用

 本文主要用于介绍创建和使用 Kubernetes Clientset

介绍

示例

[Clientset 对象](#)

## 介绍

Clientset 是调用 Kubernetes 资源对象最常用的客户端，可以操作所有的资源对象。

前面我们说了在 [staging/src/k8s.io/api](#) 下面定义了各种类型资源的规范，然后将这些规范注册到了全局的 Scheme 中，这样就可以在 [Clientset](#) 中使用这些资源了。那么我们应该如何使用 Clientset 呢？

## 示例

首先我们来看下如何通过 Clientset 来获取资源对象，我们这里来创建一个 Clientset 对象，然后通过该对象来获取默认命名空间之下的 Deployments 列表，代码如下所示：

```
package main

import (
    "flag"
    "fmt"
    "os"
    "path/filepath"

    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/rest"
    "k8s.io/client-go/tools/clientcmd"
)

func main() {
    var err error
    var config *rest.Config
    var kubeconfig *string

    if home := homeDir(); home != "" {
        kubeconfig = flag.String("kubeconfig", filepath.Join(home, ".kube", "config"), "(optional) absolute path to the kubeconfig file")
    } else {
        kubeconfig = flag.String("kubeconfig", "", "absolute path to the kubeconfig file")
    }
    flag.Parse()

    // 使用 ServiceAccount 创建集群配置 (InCluster模式)
    if config, err = rest.InClusterConfig(); err != nil {
        // 使用 KubeConfig 文件创建集群配置
        if config, err = clientcmd.BuildConfigFromFlags("", *kubeconfig); err != nil {
            panic(err.Error())
        }
    }

    // 创建 clientset
    clientset, err := kubernetes.NewForConfig(config)
    if err != nil {
        panic(err.Error())
    }

    // 使用 clientset 获取 Deployments
    deployments, err := clientset.AppsV1().Deployments("default").List(metav1.ListOptions{})
    if err != nil {
        panic(err)
    }
    for idx, deploy := range deployments.Items {
        fmt.Printf("%d -> %s\n", idx+1, deploy.Name)
    }
}
```

```
func homeDir() string {
    if h := os.Getenv("HOME"); h != "" {
        return h
    }
    return os.Getenv("USERPROFILE") // windows
}
```

上面的代码运行可以获得 default 命名空间之下的 Pods：

```
$ go run main.go
1 -> details-v1
2 -> el-gitlab-listener
3 -> nginx
4 -> productpage-v1
5 -> ratings-v1
6 -> reviews-v1
7 -> reviews-v2
8 -> reviews-v3
```

这是一个非常典型的访问 Kubernetes 集群资源的方式，通过 client-go 提供的 Clientset 对象来获取资源数据，主要有以下三个步骤：

1. 使用 kubeconfig 文件或者 ServiceAccount (InCluster 模式) 来创建访问 Kubernetes API 的 Restful 配置参数，也就是代码中的 `rest.Config` 对象
2. 使用 `rest.Config` 参数创建 Clientset 对象，这一步非常简单，直接调用 `kubernetes.NewForConfig(config)` 即可初始化
3. 然后是 Clientset 对象的方法去获取各个 Group 下面的对应资源对象进行 CRUD 操作

## Clientset 对象

上面我们了解了如何使用 Clientset 对象来获取集群资源，接下来我们来分析下 Clientset 对象的实现。

上面我们使用的 Clientset 实际上是对各种资源类型的 Clientset 的一次封装：

```
// staging/src/k8s.io/client-go/kubernetes/clientset.go

// NewForConfig 使用给定的 config 创建一个新的 Clientset
func NewForConfig(c *rest.Config) (*Clientset, error) {
    configShallowCopy := *c
    if configShallowCopy.RateLimiter == nil && configShallowCopy.QPS > 0 {
        configShallowCopy.RateLimiter = flowcontrol.NewTokenBucketRateLimiter(configShallowCopy.QPS, configShallowCopy.Burst)
    }
    var cs Clientset
    var err error
    cs.admissionregistrationV1beta1, err = admissionregistrationv1beta1.NewForConfig(&configShallowCopy)
    if err != nil {
        return nil, err
    }
    // 将其他 Group 和版本的资源的 RESTClient 封装到全局的 Clientset 对象中
    cs.appsV1, err = appsv1.NewForConfig(&configShallowCopy)
    if err != nil {
        return nil, err
    }
    .....
    cs.DiscoveryClient, err = discovery.NewDiscoveryClientForConfig(&configShallowCopy)
    if err != nil {
        return nil, err
    }
    return &cs, nil
}
```

上面的 `NewForConfig` 函数里面就是将其他的各种资源的 `RESTClient` 封装到了全局的 `Clientset` 中，这样当我们需要访问某个资源的时候只需要使用 `Clientset` 里面包装的属性即可，比如 `clientset.CoreV1()` 就是访问 Core 这个 Group 下面 v1 这个版本的 `RESTClient`。这些局部的 `RESTClient` 都定义在 `staging/src/k8s.io/client-go/typed/<group>/<version>/<plug>_client.go` 文件中，比如 `staging/src/k8s.io/client-go/kubernetes/typed/apps/v1/apps_client.go` 这个文件中就是定义的 apps 这个 Group 下面的 v1 版本的 `RESTClient`，这里同样以 Deployment 为例：

```
// staging/src/k8s.io/client-go/kubernetes/typed/apps/v1/apps_client.go
// NewForConfig 根据 rest.Config 创建一个 AppsV1Client
func NewForConfig(c *rest.Config) (*AppsV1Client, error) {
    config := *c
```

```

// 为 rest.Config 设置资源对象默认的参数
if err := setConfigDefaults(&config); err != nil {
    return nil, err
}
// 实例化 AppsV1Client 的 RestClient
client, err := rest.RESTClientFor(&config)
if err != nil {
    return nil, err
}
return &AppsV1Client{client}, nil
}

func setConfigDefaults(config *rest.Config) error {
    // 资源对象的 GroupVersion
    gv := v1.SchemeGroupVersion
    config.GroupVersion = &gv
    // 资源对象的 root path
    config.APIPath = "/apis"
    // 使用注册的资源类型 Scheme 对请求和响应进行编解码, Scheme 就是前文中分析的资源类型的规范
    config.NegotiatedSerializer = serializer.DirectCodecFactory{CodecFactory: scheme.Codecs}

    if config.UserAgent == "" {
        config.UserAgent = rest.DefaultKubernetesUserAgent()
    }

    return nil
}

func (c *AppsV1Client) Deployments(namespace string) DeploymentInterface {
    return newDeployments(c, namespace)
}

// staging/src/k8s.io/client-go/kubernetes/typed/apps/v1/deployment.go
// deployments 实现了 DeploymentInterface 接口
type deployments struct {
    client rest.Interface
    ns      string
}

// newDeployments 实例化 deployments 对象
func newDeployments(c *AppsV1Client, namespace string) *deployments {
    return &deployments{
        client: c.RESTClient(),
        ns:     namespace,
    }
}

```

通过上面代码我们就可以很清晰的知道可以通过 `clientset.AppsV1().Deployments("default")` 来获取一个 deployments 对象, 然后该对象下面定义了 deployments 对象的 CRUD 操作, 比如我们调用的 `List()` 函数:

```

// staging/src/k8s.io/client-go/kubernetes/typed/apps/v1/deployment.go

func (c *deployments) List(opts metav1.ListOptions) (result *v1.DeploymentList, err error) {
    var timeout time.Duration
    if opts.TimeoutSeconds != nil {
        timeout = time.Duration(*opts.TimeoutSeconds) * time.Second
    }
    result = &v1.DeploymentList{}
    err = c.client.Get().
        Namespace(c.ns).
        Resource("deployments").
        VersionedParams(&opts, scheme.ParameterCodec).
        Timeout(timeout).
        Do().
        Into(result)
    return
}

```

从上面代码可以看出最终是通过 `c.client` 去发起的请求, 也就是局部的 `restClient` 初始化的函数中通过 `rest.RESTClientFor(&config)` 创建的对象, 也就是将 `rest.Config` 对象转换成一个 Restful 的 Client 对象用于网络操作:

```

// staging/src/k8s.io/client-go/rest/config.go

// RESTClientFor 返回一个满足客户端 Config 对象上的属性的 RESTClient 对象。
// 注意在初始化客户端的时候, RESTClient 可能需要一些可选的属性。
func RESTClientFor(config *Config) (*RESTClient, error) {
    if config.GroupVersion == nil {
        return nil, fmt.Errorf("GroupVersion is required when initializing a RESTClient")
    }
    if config.NegotiatedSerializer == nil {

```

```

    return nil, fmt.Errorf("NegotiatedSerializer is required when initializing a RESTClient")
}
qps := config.QPS
if config.QPS == 0.0 {
    qps = DefaultQPS
}
burst := config.Burst
if config.Burst == 0 {
    burst = DefaultBurst
}

baseURL, versionedAPIPath, err := defaultServerUrlFor(config)
if err != nil {
    return nil, err
}

transport, err := TransportFor(config)
if err != nil {
    return nil, err
}
// 初始化一个 HTTP Client 对象
var httpClient *http.Client
if transport != http.DefaultTransport {
    httpClient = &http.Client{Transport: transport}
    if config.Timeout > 0 {
        httpClient.Timeout = config.Timeout
    }
}

return NewRESTClient(baseURL, versionedAPIPath, config.ContentConfig, qps, burst, config.RateLimiter, httpClient)
}

```

到这里我们就知道了 **Clientset 是基于 RESTClient 的**，RESTClient 是底层的用于网络请求的对象，可以直接通过 RESTClient 提供的 RESTful 方法如 Get()、Put()、Post()、Delete() 等和 APIServer 进行交互

- 同时支持 JSON 和 protobuf 两种序列化方式
- 支持所有原生资源

当初始化 RESTClient 过后就可以发起网络请求了，比如对于 Deployments 的 List 操作：

```

// staging/src/k8s.io/client-go/kubernetes/typed/apps/v1/deployment.go
func (c *Deployments) List(ctx context.Context, opts metav1.ListOptions) (result *v1.DeploymentList, err error) {
    var timeout time.Duration
    if opts.TimeoutSeconds != nil {
        timeout = time.Duration(*opts.TimeoutSeconds) * time.Second
    }
    result = &v1.DeploymentList{}
    // 调用 RestClient 发起真正的网络请求
    err = c.client.Get().
        Namespace(c.ns).
        Resource("deployments").
        VersionedParams(&opts, scheme.ParameterCodec).
        Timeout(timeout).
        Do(ctx).
        Into(result)
    return
}

```

上面通过调用 RestClient 发起网络请求，真正发起网络请求的代码如下所示：

```

// staging/src/k8s.io/client-go/rest/request.go
// request connects to the server and invokes the provided function when a server response is
// received. It handles retry behavior and up front validation of requests. It will invoke
// fn at most once. It will return an error if a problem occurred prior to connecting to the
// server - the provided function is responsible for handling server errors.
func (r *Request) request(ctx context.Context, fn func(*http.Request, *http.Response)) error {
    //Metrics for total request latency
    start := time.Now()
    defer func() {
        metrics.RequestLatency.Observe(r.verb, r.finalURLTemplate(), time.Since(start))
    }()

    if r.err != nil {
        klog.V(4).Infof("Error in request: %v", r.err)
        return r.err
    }

    if err := r.requestPreflightCheck(); err != nil {
        return err
    }
}

```

```

}
// 初始化网络客户端
client := r.c.Client
if client == nil {
    client = http.DefaultClient
}

// Throttle the first try before setting up the timeout configured on the
// client. We don't want a throttled client to return timeouts to callers
// before it makes a single request.
if err := r.tryThrottle(ctx); err != nil {
    return err
}
// 超时处理
if r.timeout > 0 {
    var cancel context.CancelFunc
    ctx, cancel = context.WithTimeout(ctx, r.timeout)
    defer cancel()
}

// 重试机制, 最多重试10次
maxRetries := 10
retries := 0
for {

    url := r.URL().String()
    // 构造请求对象
    req, err := http.NewRequest(r.verb, url, r.body)
    if err != nil {
        return err
    }
    req = req.WithContext(ctx)
    req.Header = r.headers

    r.backoff.Sleep(r.backoff.CalculateBackoff(r.URL()))
    if retries > 0 {
        // We are retrying the request that we already send to apiserver
        // at least once before.
        // This request should also be throttled with the client-internal rate limiter.
        if err := r.tryThrottle(ctx); err != nil {
            return err
        }
    }
    // 发起网络调用
    resp, err := client.Do(req)
    updateURLMetrics(r, resp, err)
    if err != nil {
        r.backoff.UpdateBackoff(r.URL(), err, 0)
    } else {
        r.backoff.UpdateBackoff(r.URL(), err, resp.StatusCode)
    }
    if err != nil {
        // "Connection reset by peer" or "apiserver is shutting down" are usually a transient errors.
        // Thus in case of "GET" operations, we simply retry it.
        // We are not automatically retrying "write" operations, as
        // they are not idempotent.
        if r.verb != "GET" {
            return err
        }
    }
    // For connection errors and apiserver shutdown errors retry.
    if net.IsConnectionReset(err) || net.IsProbableEOF(err) {
        // For the purpose of retry, we set the artificial "retry-after" response.
        // TODO: Should we clean the original response if it exists?
        resp = &http.Response{
            StatusCode: http.StatusInternalServerError,
            Header:      http.Header{"Retry-After": []string{"1"}},
            Body:        ioutil.NopCloser(bytes.NewReader([]byte{})),
        }
    } else {
        return err
    }
}

done := func() bool {
    // Ensure the response body is fully read and closed
    // before we reconnect, so that we reuse the same TCP
    // connection.
    defer func() {
        const maxBodySlurpSize = 2 << 10
        if resp.ContentLength <= maxBodySlurpSize {
            io.Copy(ioutil.Discard, &io.LimitedReader{R: resp.Body, N: maxBodySlurpSize})
        }
        resp.Body.Close()
    }()

    retries++
}

```

```

    if seconds, wait := checkWait(resp); wait && retries < maxRetries {
        if seeker, ok := r.body.(io.Seeker); ok && r.body != nil {
            _, err := seeker.Seek(0, 0)
            if err != nil {
                klog.V(4).Infof("Could not retry request, can't Seek() back to beginning of body for %T", r.body)
                fn(req, resp)
                return true
            }
        }
    }

    klog.V(4).Infof("Got a Retry-After %ds response for attempt %d to %v", seconds, retries, url)
    r.backoff.Sleep(time.Duration(seconds) * time.Second)
    return false
}
// 将req和resp回调
fn(req, resp)
return true
}()
if done {
    return nil
}
}
}

```

到这里就完成了完整的网络请求。

其实 Clientset 对象也就是将 `rest.Config` 封装成了一个 `http.Client` 对象而已，最终还是利用 `golang` 中的 `http` 库来执行一个正常的网络请求而已。