

Если вы видите что-то необычное, просто сообщите мне.

Доступ к K8S CRD из go-клиента

Kubernetes API сервер легко расширяется с помощью Custom Resource Defenition. Однако, доступ к этом ресурсу из популярных библиотек go-клиентов сложна и плохо задокументированна. Эта статья содержит маленькую инструкцию как получить доступ к такому ресурсу из вашего кода Go.

Цель

Я пришел к этой задаче, когда хотел интегрировать внешнее хранилище в кубернетес кластер. План был использовать CRD, что бы определять резервы систем хранения данных. Потом, самодельный оператор может слушать все эти ресурсы чтобы создавать и удалять и управлять текущим состоянием этих ресурсов.

Определим и создадим CRD

Для этой статьи будем работать над простым примером: CRD может быть леко создан используя `kubectl` для этого примера, мы начнем с одиночного простого определения ресурса:

```
apiVersion: "apiextensions.k8s.io/v1beta1"
kind: "CustomResourceDefinition"
metadata:
```

```
name: "projects.example.martin-helmich.de"
spec:
  group: "example.martin-helmich.de"
  version: "v1alpha1"
  scope: "Namespaced"
  names:
    plural: "projects"
    singular: "project"
    kind: "Project"
  validation:
    openAPIV3Schema:
      required: ["spec"]
      properties:
        spec:
          required: ["replicas"]
          properties:
            replicas:
              type: "integer"
              minimum: 1
```

Для определения CRD, нам понадобится озаботиться об API Group Name(в этом случае, `example.martin-helmich.de`). По соглашению, это обычно доменное, которым вы владеете(например домен организации), чтобы предотвратить конфликты наименования. CRD имена обычно выглядят так: `<plural-resource-name>.<api-group-name>`, в нашем примере: `projects.example.martin-helmich.de` .

Так же, будьте внимательны когда выбираете версию CRD(`spec.version` в примере выше). Пока еще работает над CRD, то будет хорошей идеей поместить CRD в группу alpha версии API. Для пользователей вашего самодельного ресурса, это будет значить, что что-то может измениться.

Часто, нужно проверить что данные которые хранит пользователь в вашем CRD содержит определенные схемы. За это отвечает `spec.validation.openAPIV3Schema` . Она содержит JSON схему которая описывает формат который должны иметь CRD.

После сохранения CRD в файл, применим его в кластере:

```
> kubectl apply -f projects-crd.yaml
customresourcedefinition "projects.example.martin-helmich.de" created
```

После создания CRD вы можете создать объект этого типа. Работает это так же как с обычными Kubernetes Объектами(pods, deployments и так далее). Отличается только `kind` и `apiVersion`:

```
apiVersion: "example.martin-helmich.de/v1alpha1"
kind: "Project"
metadata:
  name: "example-project"
  namespace: "default"
spec:
  replicas: 1
```

Можно создать CRD как любой другой объект через `kubectl`

```
> kubectl apply -f project.yaml
project "example-project" created
```

Можно даже использовать `kubectl` чтобы получить самодельный ресурс обратно из K8S.

```
> kubectl get projects
NAME          AGE
example-project 2m
```

Создание Golang клиента

Теперь, будем использовать пакет го-клиента, для доступа к этим CRD. Для примера, я буду считать, что мы работаем над Go проектом, с названием `github.com/martin-helmich/kubernetes-crd-example` (репозиторий существует) и у него есть го-клиент и `apimachinery` установленную библиотеку в качестве модуля Go.

```
go mod init github.com/martin-helmich/kubernetes-crd-example
go get k8s.io/client-go@v0.17.0
go get k8s.io/apimachinery@v0.17.0
```

Множество документаций работая с CRD предполагают, что вы работаете с некоторым типом генерации кода, чтобы собрать клиентскую библиотеку автоматически. Однако, этот процесс задокументирован редко, и после прочтения нескольких ненавистных дискуссий на github, создалось впечатление, что всё ещё в "прогрессе". В общем, я встал в самостоятельную реализацию клиентат.

Шаг 1: Определение типов

Начав с определения типов для самодельного ресурса. Я нашел, что это хорошая практика, организовывать эти типа группируя по версии API. Для примера, можно создать файл `api/types/v1alpha1/project.go` содержащий следующее:

```
package v1alpha1

import metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"

type ProjectSpec struct {
    Replicas int `json:"replicas"`
}

type Project struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec ProjectSpec `json:"spec"`
}

type ProjectList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata,omitempty"`

    Items []Project `json:"items"`
}
```

Тип `metav1.ObjectMeta` содержит типичные свойства метадаты, которые вы можете найти в любом K8s ресурсе.

Шаг 2: Определим метод DeepCopy

Каждый тип, который будет обслуживаться K8S API(в нашем случае, Project и ProjectList) требует реализацию `k8s.io/apimachinery/pkg/runtime.Object` интерфейса. Этот интерфейс определяет 2 метода `GetObjectKind()` и `DeepCopyObject()`. Первый метод уже предоставлен встроенной структурой `metav1.TypeMeta`, второй нужно реализовать самостоятельно.

Метод `DeepCopyObject` предназначен для создания полной копии объекта. Так как это требует шаблонного кода, этот метод часто генерируется автоматически. Для этой статьи мы сделаем это ручками. Продолжим с добавления второго файла `deepcopy.go` в тот же пакет:

```
package v1alpha1

import "k8s.io/apimachinery/pkg/runtime"

// DeepCopyInto copies all properties of this object into another object of the
// same type that is provided as a pointer.
func (in *Project) DeepCopyInto(out *Project) {
    out.TypeMeta = in.TypeMeta
    out.ObjectMeta = in.ObjectMeta
    out.Spec = ProjectSpec{
        Replicas: in.Spec.Replicas,
    }
}

// DeepCopyObject returns a generically typed copy of an object
func (in *Project) DeepCopyObject() runtime.Object {
    out := Project{}
    in.DeepCopyInto(&out)

    return &out
}

// DeepCopyObject returns a generically typed copy of an object
```

```
func (in *ProjectList) DeepCopyObject() runtime.Object {
    out := ProjectList{}
    out.TypeMeta = in.TypeMeta
    out.ListMeta = in.ListMeta

    if in.Items != nil {
        out.Items = make([]Project, len(in.Items))
        for i := range in.Items {
            in.Items[i].DeepCopyInto(&out.Items[i])
        }
    }

    return &out
}
```

Интерлюдия: Автоматическое создание DeepCopy метода

Так, мы могли заметить, что определение всех этих различных DeepCopy методов вовсе не веселое занятие. Есть множество различных инструментов и фреймворков около автогенерации этих методов(все зависит от уровня документации и в целом зрелости). То что нашел я, работает отлично в инструменте `controller-gen`, что является частью фреймворка `Kubebuilder`:

```
$ go get -u github.com/kubernetes-sigs/controller-tools/cmd/controller-gen
```

Чтобы использовать `controller-gen`, опишите ваш CRD тип через `+k8s:deepcopy-gen annotation`

```
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
type Project struct {
    // ...
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
type ProjectList struct {
```

```
// ...  
}
```

Затем, выполните команду, для автоматического создания метода `deerpory`

```
controller-gen object paths=./api/types/v1alpha1/project.go
```

Можно еще проще, вы можете добавить `go:generate` выражение в целый файл:

```
package v1alpha1  
  
import metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"  
  
//go:generate controller-gen object paths=$GOFILE  
  
// ...
```

И чтобы сгенерировать код, нужно выполнить команду в корневой папке:

```
go generate ./...
```

Шаг 3: Зарегистрируем типы на схеме компоновщика

Теперь, нам нужно сделать новый тип известным для библиотеки клиента. Это позволить клиенту(более или менее) автоматически обрабатывать ваши новые типы после подключения к API серверу.

Для этого, добавим новый файл `register.go` в наш пакет:

```
package v1alpha1  
  
import (  
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"  
    "k8s.io/apimachinery/pkg/runtime"  
    "k8s.io/apimachinery/pkg/runtime/schema"
```

```

)

const GroupName = "example.martin-helmich.de"
const GroupVersion = "v1alpha1"

var SchemeGroupVersion = schema.GroupVersion{Group: GroupName, Version: GroupVersion}

var (
    SchemeBuilder = runtime.NewSchemeBuilder(addKnownTypes)
    AddToScheme   = SchemeBuilder.AddToScheme
)

func addKnownTypes(scheme *runtime.Scheme) error {
    scheme.AddKnownTypes(SchemeGroupVersion,
        &Project{},
        &ProjectList{},
    )

    metav1.AddToGroupVersion(scheme, SchemeGroupVersion)
    return nil
}

```

Как можно заметить, этот код не делает что-то реальное, пока еще(за исключением создания нового `runtime.SchemeBuilder`). Важная часть в том, что `AddToScheme` функция(строка 16)б которая экспортирует членов структуры созданных с типом `runtime.SchemeBuilder` в строке 15. Вы можете вызвать эту функцию позже, из любой части вашего клиентского кода как только клиент K8S будет готов к регистрации вашего определенного типа.

Шаг 4: создание HTTP клиента

После определения типов и добавления метода для регистрации их в глобальной схеме компоновщика, вы можете создат HTTP клиента, который может загружать ваши собственные ресурсы.

Для этого, добавим следующий код в ваш `main.go` вашего пакета


```
package main

import (
    "flag"
    "log"
    "time"

    "k8s.io/apimachinery/pkg/runtime/schema"
    "k8s.io/apimachinery/pkg/runtime/serializer"

    "github.com/martin-helmich/kubernetes-crd-example/api/types/v1alpha1"
    "k8s.io/client-go/kubernetes/scheme"
    "k8s.io/client-go/rest"
    "k8s.io/client-go/tools/clientcmd"
)

var kubeconfig string

func init() {
    flag.StringVar(&kubeconfig, "kubeconfig", "", "path to Kubernetes config file")
    flag.Parse()
}

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

    if kubeconfig == "" {
        log.Printf("using in-cluster configuration")
        config, err = rest.InClusterConfig()
    } else {
        log.Printf("using configuration from '%s'", kubeconfig)
        config, err = clientcmd.BuildConfigFromFlags("", kubeconfig)
    }

    if err != nil {
        panic(err)
    }

    v1alpha1.AddToScheme(scheme.Scheme)
```

```

crdConfig := *config
crdConfig.ContentConfig.GroupVersion = &schema.GroupVersion{Group: v1alpha1.GroupName, Version:
v1alpha1.GroupVersion}
crdConfig.APIPath = "/apis"
crdConfig.NegotiatedSerializer = serializer.NewCodecFactory(scheme.Scheme)
crdConfig.UserAgent = rest.DefaultKubernetesUserAgent()

exampleRestClient, err := rest.UnversionedRESTClientFor(&crdConfig)
if err != nil {
    panic(err)
}
}

```

Теперь можно использовать `exampleRestClient` созданный в строке 48, для запроса всех самостоятельных ресурсов внутри `example.martin-helmich.de/v1alpha1` API группы. Пример может выглядеть следующим образом:

```

result := v1alpha1.ProjectList{}
err := exampleRestClient.
    Get().
    Resource("projects").
    Do().
    Into(&result)

```

Чтобы использовать API типобезопасным способом, обычно хорошая идея обернуть эти операции внутри своего клиентского набора. Для этого, создаём новый подпакет `clientset/v1alpha1`. Для начала, реализуем интерфейс который определяет типы для вашей группы API. И Переносим настройки конфигурации из вашего главного метода в эту функцию конструктора клиентского набора(`NewForConfig` для примера ниже):

```

package v1alpha1

import (
    "github.com/martin-helmich/kubernetes-crd-example/api/types/v1alpha1"
    "k8s.io/apimachinery/pkg/runtime/schema"
    "k8s.io/apimachinery/pkg/runtime/serializer"
    "k8s.io/client-go/kubernetes/scheme"
    "k8s.io/client-go/rest"

```

```

)

type ExampleV1Alpha1Interface interface {
    Projects(namespace string) ProjectInterface
}

type ExampleV1Alpha1Client struct {
    restClient rest.Interface
}

func NewForConfig(c *rest.Config) (*ExampleV1Alpha1Client, error) {
    config := *c
    config.ContentConfig.GroupVersion = &schema.GroupVersion{Group: v1alpha1.GroupName, Version:
v1alpha1.GroupVersion}
    config.APIPath = "/apis"
    config.NegotiatedSerializer = scheme.Codecs.WithoutConversion()
    config.UserAgent = rest.DefaultKubernetesUserAgent()

    client, err := rest.RESTClientFor(&config)
    if err != nil {
        return nil, err
    }

    return &ExampleV1Alpha1Client{restClient: client}, nil
}

func (c *ExampleV1Alpha1Client) Projects(namespace string) ProjectInterface {
    return &projectClient{
        restClient: c.restClient,
        ns: namespace,
    }
}

```

Код ниже, всё еще, не будет компилироваться, так как в нем всё еще отсутствуют `ProjectInterface` и `projectClient` типы. Мы сейчас до них доберемся.

`ExampleV1Alpha1Interface` и его реализация, `ExampleV1Alpha1Client` структура это главная точка входа для доступа к самодельным ресурсам. Вы можете легко создать новый клиентский набор в вашем `main.go`, просто вызывая `clientset, err := v1alpha1.NewForConfig(config)`.

Дальше, вам нужно реализовать определенный клиентский набор для доступа к самодельному ресурсу `Project` (пример выше уже использует `ProjectInterface` и `projectClient` типы которые всё еще нужно поддерживать). Создадим второй файл в том же пакете `projects.go`:

```
package v1alpha1

import (
    "github.com/martin-helmich/kubernetes-crd-example/api/types/v1alpha1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/watch"
    "k8s.io/client-go/kubernetes/scheme"
    "k8s.io/client-go/rest"
)

type ProjectInterface interface {
    List(opts metav1.ListOptions) (*v1alpha1.ProjectList, error)
    Get(name string, options metav1.GetOptions) (*v1alpha1.Project, error)
    Create(*v1alpha1.Project) (*v1alpha1.Project, error)
    Watch(opts metav1.ListOptions) (watch.Interface, error)
    // ...
}

type projectClient struct {
    restClient rest.Interface
    ns         string
}

func (c *projectClient) List(opts metav1.ListOptions) (*v1alpha1.ProjectList, error) {
    result := v1alpha1.ProjectList{}
    err := c.restClient.
        Get().
        Namespace(c.ns).
        Resource("projects").
        VersionedParams(&opts, scheme.ParameterCodec).
        Do().
        Into(&result)

    return &result, err
}
```

```

func (c *projectClient) Get(name string, opts metav1.GetOptions) (*v1alpha1.Project, error) {
    result := v1alpha1.Project{}
    err := c.restClient.
        Get().
        Namespace(c.ns).
        Resource("projects").
        Name(name).
        VersionedParams(&opts, scheme.ParameterCodec).
        Do().
        Into(&result)

    return &result, err
}

func (c *projectClient) Create(project *v1alpha1.Project) (*v1alpha1.Project, error) {
    result := v1alpha1.Project{}
    err := c.restClient.
        Post().
        Namespace(c.ns).
        Resource("projects").
        Body(project).
        Do().
        Into(&result)

    return &result, err
}

func (c *projectClient) Watch(opts metav1.ListOptions) (watch.Interface, error) {
    opts.Watch = true
    return c.restClient.
        Get().
        Namespace(c.ns).
        Resource("projects").
        VersionedParams(&opts, scheme.ParameterCodec).
        Watch()
}

```

Этот клиент очевидно еще не закончен и не имеет методы типа `Delete`, `Update` и другие. Однако, это можно реализовать похожим на существующий метод образом. Посмотрите на

существующий клиентский набор(для примера `Pod client set`) для вдохновения.

После создания вашего клиентского набора и используя его, вывести список существующих ресурсов становится довольно легко.

```
import clientV1alpha1 "github.com/martin-helmich/kubernetes-crd-example/clientset/v1alpha1"
// ...

func main() {
    // ...

    clientSet, err := clientV1alpha1.NewForConfig(config)
    if err != nil {
        panic(err)
    }

    projects, err := clientSet.Projects("default").List(metav1.ListOptions{})
    if err != nil {
        panic(err)
    }

    fmt.Printf("projects found: %+v\n", projects)
}
```

Шаг 5: Создаем оповещатель

При создании оператора Kubernetes. Вы обычно хотите иметь возможность реагировать на вновь созданные или обновленные ресурсы. В теории, вы можете просто периодически вызывать `List()` метод и проверять добавлены ли новые ресурсы. На практике, это не оптимальное решение, особенно когда у вас есть множество подобных ресурсов.

Большинство операторов работает изначально загрузив все актуальные экземпляры ресурсов используя начальный вызов `List()`, и затем подписываясь на обновления с помощью `Watch()` вызова. Начальный список объектов и обновления полученные от `Watch()` далее используются для создания локального кэша, что позволяет иметь быстрый доступ к любому самодельному ресурсу без надобности хождения к API серверу каждый раз.

Этот шаблон широко распространен, что библиотеки go-клиентов предлагают готовое решение: пакет `k8s.io/client-go/tools/cache`. Вы можете создать новый оповещатель для ваших ресурсов:

```
package main

import (
    "time"

    "github.com/martin-helmich/kubernetes-crd-example/api/types/v1alpha1"
    client_v1alpha1 "github.com/martin-helmich/kubernetes-crd-example/clientset/v1alpha1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/runtime"
    "k8s.io/apimachinery/pkg/util/wait"
    "k8s.io/apimachinery/pkg/watch"
    "k8s.io/client-go/tools/cache"
)

func WatchResources(clientSet client_v1alpha1.ExampleV1Alpha1Interface) cache.Store {
    projectStore, projectController := cache.NewInformer(
        &cache.ListWatch{
            ListFunc: func(lo metav1.ListOptions) (result runtime.Object, err error) {
                return clientSet.Projects("some-namespace").List(lo)
            },
            WatchFunc: func(lo metav1.ListOptions) (watch.Interface, error) {
                return clientSet.Projects("some-namespace").Watch(lo)
            },
        },
        &v1alpha1.Project{},
        1*time.Minute,
        cache.ResourceEventHandlerFuncs{},
    )

    go projectController.Run(wait.NeverStop)
    return projectStore
}
```

Метод `NewInformer` возвращает два объекта: второй - значение, `controller` - управляет `List()` и `Watch()` вызывает и наполняет первое значение, храня некоторое количество кэшированных

ресурсов с API сервера(в нашем случае CRD).

Теперь можно использовать хранилище, для легкого доступа к вашему CRD, либо слушая их или иметь доступ к ним по именам. Помните, что функции хранения возвращают `interface` типа, поэтому вам нужно будет самостоятельно приводить их к CRD типам.

```
store := WatchResource(clientSet)
```

```
project := store.GetByKey("some-namespace/some-project").(*v1alpha1.Project)
```

Вывод

Создание клиентов для CRD - это что-то что мало задокументированно(на данный момент) и подчас может быть довольно сложным.

Клиентская библиотека для CRD, что показана в статье, вместе с оповещателем это отличный старт для создания вашего собственного K8S оператора который реагирует на изменения который делают CRD.

Revision #2

Created 21 March 2022 13:11:04 by gasick

Updated 16 April 2023 19:30:03 by gasick