K8s管理系统项目实战【API开发】

> 讲师：杜Sir
>
> 阿良教育： www.aliangedu.cn

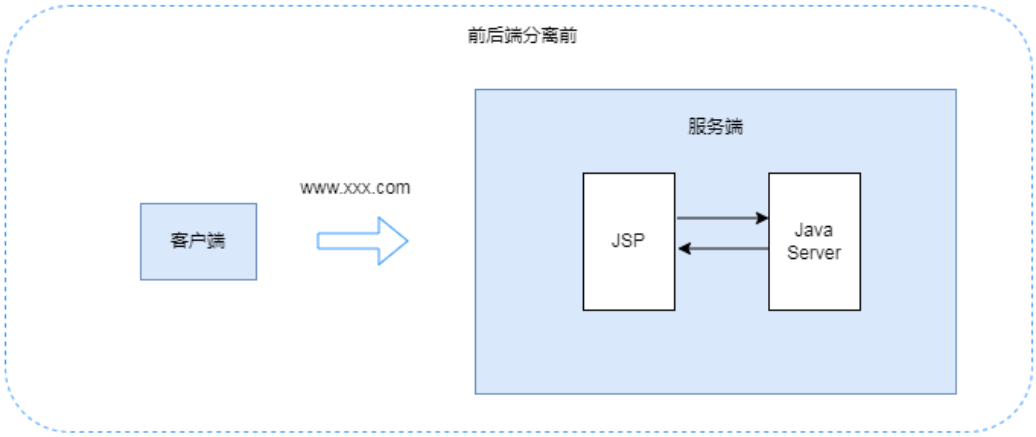
一、项目背景

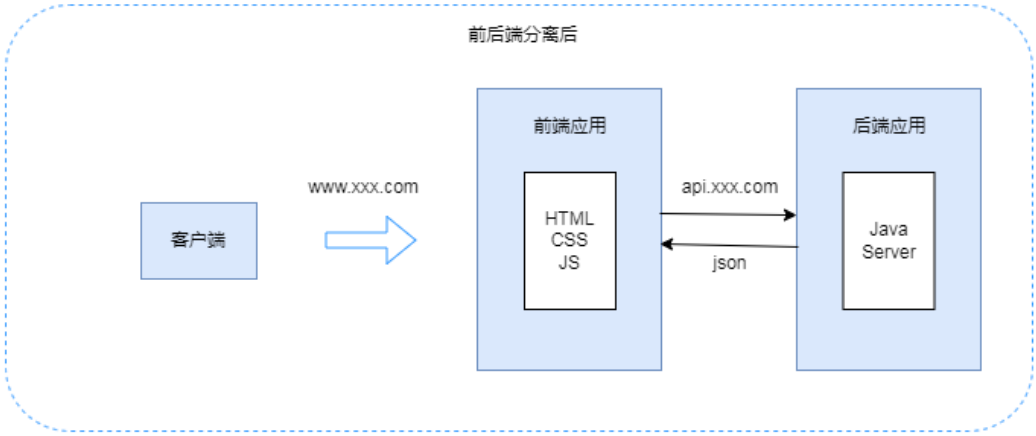
随着容器技术的广泛应用，kubernetes逐渐成为业内的核心技术，是容器编排技术的首选工具。而k8s管理平台在日常的容器维护中也发挥着举足轻重的作用，但随着k8s的定制化功能越来越多，dashboard已经无法满足日常的维护需求，且dashboard的源码学习成本较高，抽象程度较高，二次开发成本也就比较高。

本项目使用当下较主流的前端vue+element plus及后端go+gin框架，打造与dashboard对标的k8s管理功能，且可定制化程度高，同学们在入门后可根据自身需求，进行灵活定制开发。且本课程除了学习相关知识点外，会带着大家逐渐掌握开发习惯与技巧，让大家在日常工作中高效、高质量的进行开发作业。

二、前后端分离概述

前后端分离已成为互联网项目开发的业界标准使用方式，通过nginx+tomcat的方式（也可以中间加一个nodejs）有效的进行解耦，并且前后端分离会为以后的大型分布式架构、弹性计算架构、微服务架构、l多端化服务（多种客户端，例如：浏览器，车载终端，安卓，IOS等等）打下坚实的基础。这个步骤是系统架构从猿进化成人的必经之路。

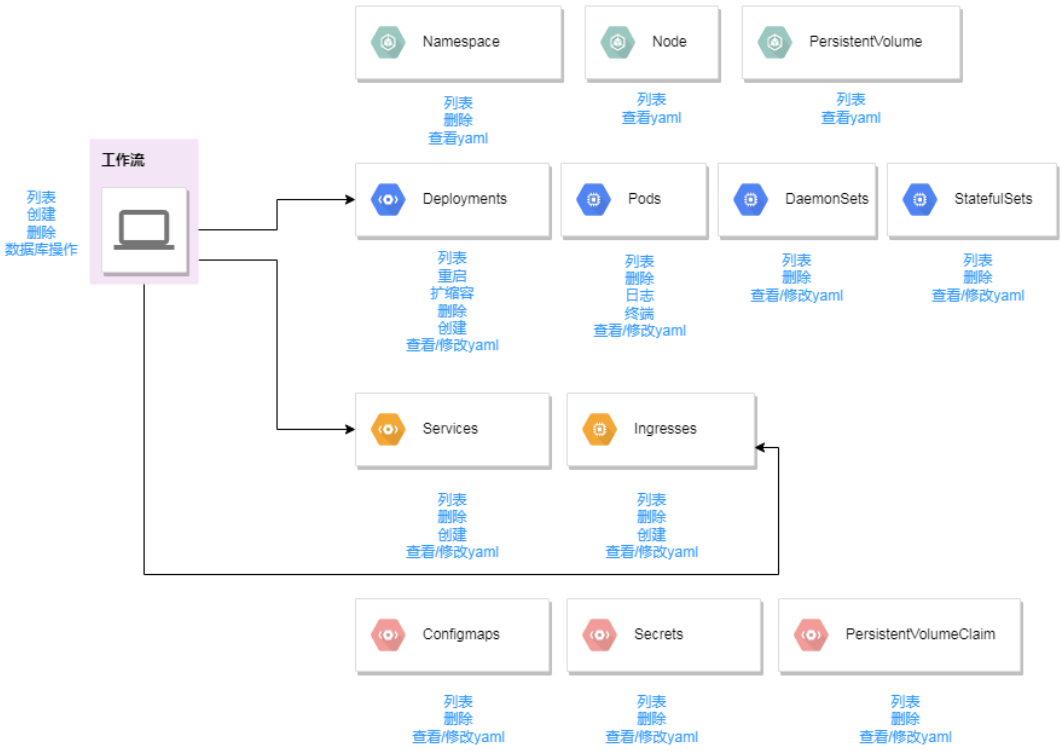




前后分离的优势：

1. 可以实现真正的前后端解耦，前端服务器使用nginx。
2. 发现bug，可以快速定位是谁的问题，不会出现互相踢皮球的现象。
3. 在大并发情况下，可以同时水平扩展前后端服务器。
4. 增加代码的维护性&易读性（前后端耦在一起的代码读起来相当费劲）。
5. 提升开发效率，因为可以前后端并行开发，而不是像以前的强依赖。

三、功能设计



四、Client-go介绍

1、简介

client-go是kubernetes官方提供的go语言的客户端库，go应用使用该库可以访问kubernetes的API Server，这样我们就能通过编程来对kubernetes资源进行增删改查操作；

除了提供丰富的API用于操作kubernetes资源，client-go还为controller和operator提供了重要支持client-go的informer机制可以将controller关注的资源变化及时带给此controller，使controller能够及时响应变化。

通过client-go提供的客户端对象与kubernetes的API Server进行交互，而client-go提供了以下四种客户端对象：

（1）RESTClient：这是最基础的客户端对象，仅对HTTPRequest进行了封装，实现RESTFul风格API，这个对象的使用并不方便，因为很多参数都要使用者来设置，于是client-go基于RESTClient又实现了三种新的客户端对象；

（2）ClientSet：把Resource和Version也封装成方法了，用起来更简单直接，一个资源是一个客户端，多个资源就对应了多个客户端，所以ClientSet就是多个客户端的集合了，这样就好理解了，不过ClientSet只能访问内置资源，访问不了自定义资源；

（3）DynamicClient：可以访问内置资源和自定义资源，个人感觉有点像java的集合操作，拿出的内容是Object类型，按实际情况自己去做强制转换，当然了也会有强转失败的风险；

（4）DiscoveryClient：用于发现kubernetes的API Server支持的Group、Version、Resources等信息；

//kubectl api-resources

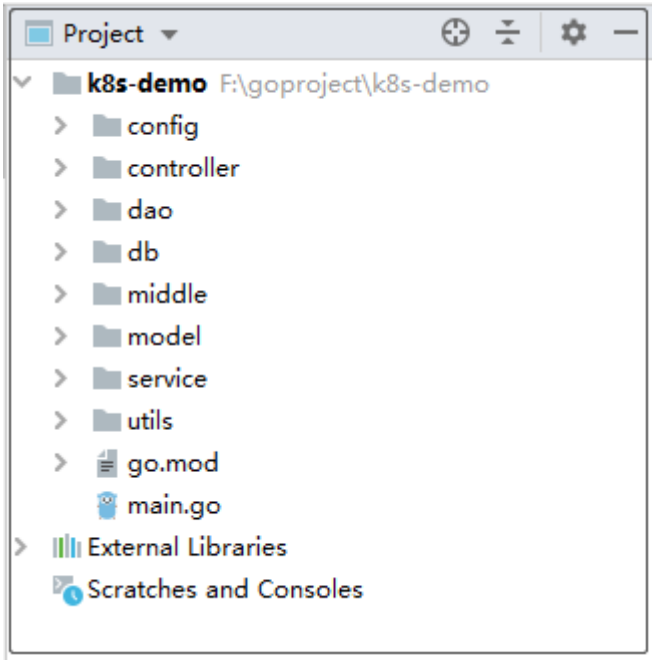
2、代码示例

```
package main

import (
    "context"
    "flag"
    "fmt"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/tools/clientcmd"
    "os"
    "path/filepath"
)

func main() {
    //声明kubeconfig配置文件
    var kubeconfig *string
    //获取home环境变量，拿到$HOME/.kube/config配置文件
    if home := homeDir(); home != "" {
        kubeconfig = flag.String("kubeconfig", filepath.Join(home, ".kube", "config"),
"(optional) absolute path to the kubeconfig file")
    } else {
        //否则就根据kubeconfig传到获得config的路径
        kubeconfig = flag.String("kubeconfig", "", "absolute path to the kubeconfig
file")
    }
    flag.Parse()
    //将kubeconfig格式化为rest.config类型
    config, err := clientcmd.BuildConfigFromFlags("", *kubeconfig)
    if err != nil {
        panic(err.Error())
    }
    //通过config创建clientset
    clientset, err := kubernetes.NewForConfig(config)
    if err != nil {
        panic(err.Error())
    }
    //通过client-go sdk获取pods列表，命名空间是default
    pods, err := clientset.CoreV1().Pods("default").List(context.TODO(),
metav1.ListOptions{})
    if err != nil {
        panic(err.Error())
    }
    //打印出pods列表的长度
    fmt.Printf("There are %d pods in the cluster\n", len(pods.Items))
}


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

3、常用方法

```
//获取pod列表
podList, err := K8s.ClientSet.CoreV1().Pods(namespace).List(context.TODO(),
metav1.ListOptions{})

//获取pod详情
pod, err = K8s.ClientSet.CoreV1().Pods(namespace).Get(context.TODO(), podName,
metav1.GetOptions{})

//删除pod
err = K8s.ClientSet.CoreV1().Pods(namespace).Delete(context.TODO(), podName,
metav1.DeleteOptions{})

//更新pod（完整yaml）
pod, err = K8s.ClientSet.CoreV1().Pods(namespace).Update(context.TODO(), pod,
metav1.UpdateOptions{})

//获取deployment副本数
scale, err := K8s.ClientSet.AppsV1().Deployments(namespace).GetScale(context.TODO(),
deploymentName, metav1.GetOptions{})

//创建deployment
deployment, err =
K8s.ClientSet.AppsV1().Deployments(data.Namespace).Create(context.TODO(), deployment,
metav1.CreateOptions{})

//更新deployment（部分yaml）
deployment, err = K8s.ClientSet.AppsV1().Deployments(namespace).Patch(context.TODO(),
deploymentName, "application/strategic-merge-patch+json", patchByte,
metav1.PatchOptions{})
```

五、项目目录结构



config：定义全局配置，如监听地址、管理员账号等。

controller：controller层，定义路由规则，及接口入参和响应。

service：服务层，处理接口的业务逻辑。

dao：数据库操作，包含数据库的增删改查。

model：定义数据库的表的字段。

db: 用于初始化数据库连接以及配置。

middle：中间件层，添加全局的逻辑处理，如跨域、jwt验证等。

utils：工具目录，定义常用工具，如token解析，文件操作等。

go.mod：定义项目的依赖包以及版本。

main.go：项目的主入口 main函数。

六、框架搭建

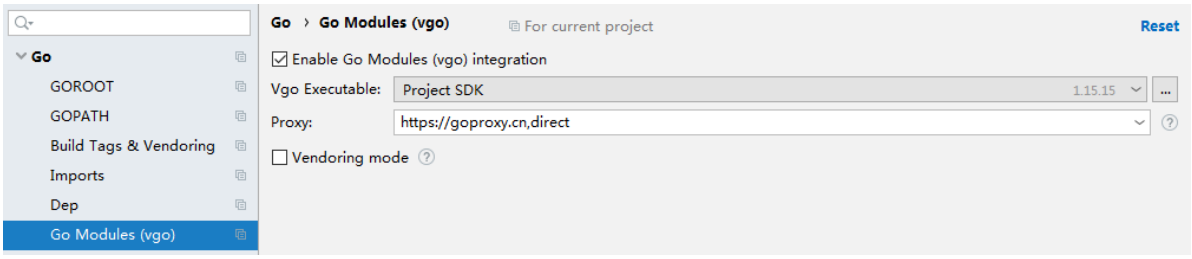
> 阿良教育： www.aliangedu.cn

1、创建Go项目

（1）新建Go项目文件夹，命名为tiga

（2）使用GoLand打开tiga目录，在tiga目录下创建main.go文件

（3）打开GoLand Settings，配置GOROOT和GO Modules（vgo）



（4）初始化go mod文件

```
go mod init tiga
```

2、初始化Gin工程

（1）创建项目目录结构（见第四节）

（2）创建config/config.go文件，配置启动监听端口

```
package config

const (
    ListenAddr = "0.0.0.0:9090"
)
```

（3）创建controller/router.go文件，初始化api路由规则，编写测试api接口

```
package controller

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

//实例化router结构体，可使用该对象点出首字母大写的方法（包外调用）
var Router router

//创建router结构体
type router struct {}

//初始化路由规则，创建测试api接口
func(r *router) InitApiRouter(router *gin.Engine) {
    router.GET("/testapi", func(ctx *gin.Context){
        ctx.JSON(http.StatusOK, gin.H{
            "msg": "testapi success!",
            "data": nil,
        })
    })
}
```

（4）main.go文件中，启动gin程序

```
package main

import (
    "github.com/gin-gonic/gin"
    "tiga/config"
    "tiga/controller"
)

func main() {
    //初始化gin对象
    r := gin.Default()
    //初始化路由规则
    controller.Router.InitApiRouter(r)
    //gin程序启动
    r.Run(config.ListenAddr)
}
```

3、初始化K8s Client

（1）在config.go中添加kubeconfig文件路径

```
package config

const (
    ListenAddr = "0.0.0.0:11010"
    Kubeconfig = "F:\\goproject\\config"
)
```

（2）创建service/init.go文件，初始化k8s client

```
package service

import (
    "github.com/wonderivan/logger"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/tools/clientcmd"
    "newpro/config"
)

var K8s k8s

type k8s struct {
    ClientSet *kubernetes.Clientset
}

func(k *k8s) Init() {
    conf, err := clientcmd.BuildConfigFromFlags("", config.Kubeconfig)
    if err != nil {
        logger.Error("创建k8s配置失败, " + err.Error())
    }

    clientSet, err := kubernetes.NewForConfig(conf)
    if err != nil {
        logger.Error("创建k8s clientSet失败, " + err.Error())
    } else {
        logger.Info("创建k8s clientSet成功")
    }

    k.ClientSet = clientSet
}
```

（3）main.go中添加k8s client启动项

```
package main

import (
    "github.com/gin-gonic/gin"
    "tiga/config"
    "tiga/controller"
)

func main() {
    //初始化gin对象
    r := gin.Default()
```

```
    //初始化k8s client
    service.K8s.Init()
    //初始化路由规则
    controller.Router.InitApiRouter(r)
    //gin程序启动
    r.Run(config.ListenAddr)
}
```

4、测试API接口

七、API开发

1、开发&响应流程

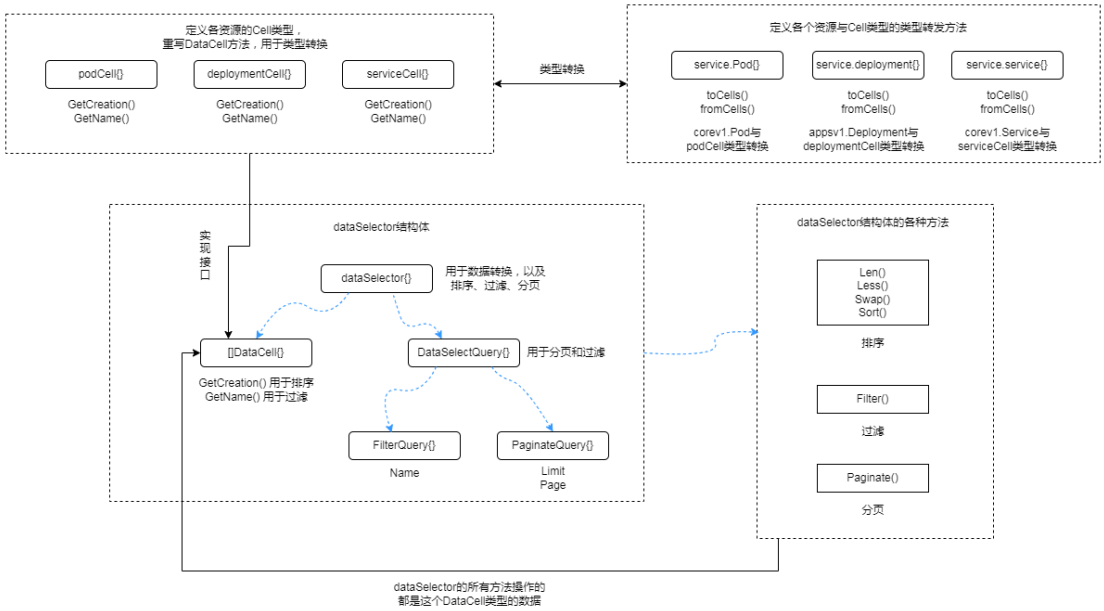
（1）Model层：实体层 >> 表数据与类的映射关系，这里的类是strcut

（2）Dao层：持久层/数据访问层 >> 主要与数据库交互，增删改查

（3）Service层：业务层 >> 控制业务逻辑，主要处理业务模块的功能逻辑

（4）Controller层：控制层 >> 接收请求参数，调用不同的Service层代码来控制业务流程

2、工作负载



2.1 数组的排序、过滤、分页

service/dataselector.go

（1）定义数据结构

```
package service

import (
    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    nwv1 "k8s.io/api/networking/v1"
    "sort"
    "strings"
    "time"
)

//dataSelect 用于封装排序、过滤、分页的数据类型
type dataSelector struct {
    GenericDataList []DataCell
    dataSelectQuery *DataSelectQuery
}
//DataCell接口，用于各种资源list的类型转换，转换后可以使用dataSelector的自定义排序方法
type DataCell interface {
    GetCreation() time.Time
    GetName()     string
}
//DataSelectQuery 定义过滤和分页的属性，过滤：Name， 分页：Limit和Page
//Limit是单页的数据条数
//Page是第几页
type DataSelectQuery struct {
    FilterQuery    *FilterQuery
    PaginateQuery  *PaginateQuery
}

type FilterQuery struct {
    Name string
}
```

```
type PaginateQuery struct {
    Limit int
    Page  int
}
```

(2) 排序

自定义类型排序参考文档： https://segmentfault.com/a/1190000008062661

```
//实现自定义结构的排序，需要重写Len、Swap、Less方法
//Len方法用于获取数组长度
func(d *dataSelector) Len() int {
    return len(d.GenericDataList)
}
//Swap方法用于数组中的元素在比较大小后的位置交换，可定义升序或降序
func(d *dataSelector) Swap(i, j int) {
    d.GenericDataList[i], d.GenericDataList[j] = d.GenericDataList[j],
d.GenericDataList[i]
}
//Less方法用于定义数组中元素排序的“大小”的比较方式
func(d *dataSelector) Less(i, j int) bool {
    a := d.GenericDataList[i].GetCreation()
    b := d.GenericDataList[j].GetCreation()

    return b.Before(a)
}
//重写以上3个方法用使用sort.Sort进行排序
func(d *dataSelector) Sort() *dataSelector {
    sort.Sort(d)
    return d
}
```

(3) 过滤

```
//Filter方法用于过滤元素，比较元素的Name属性，若包含，再返回
func(d *dataSelector) Filter() *dataSelector{
    //若Name的传参为空，则返回所有元素
    if d.dataSelectQuery.FilterQuery.Name == "" {
        return d
    }
    //若Name的传参不为空，则返回元素名中包含Name的所有元素
    filteredList := []DataCell{}
    for _, value := range d.GenericDataList {
        matches := true
        objName := value.GetName()
        if !strings.Contains(objName, d.dataSelectQuery.FilterQuery.Name) {
            matches = false
            continue
        }
        if matches {
            filteredList = append(filteredList, value)
        }
    }

    d.GenericDataList = filteredList
    return d
}
```

（4）分页

```
//Paginate方法用于数组分页，根据Limit和Page的传参，返回数据
func(d *dataSelector) Paginate() *dataSelector{
    limit := d.dataSelectQuery.PaginateQuery.Limit
    page := d.dataSelectQuery.PaginateQuery.Page
    //验证参数合法，若参数不合法，则返回所有数据
    if limit <= 0 || page <= 0 {
        return d
    }
    //举例：25个元素的数组，limit是10，page是3，startIndex是20，endIndex是30（实际上endIndex是25）
    startIndex := limit * (page - 1)
    endIndex := limit * page

    //处理最后一页，这时候就把endIndex由30改为25了
    if len(d.GenericDataList) < endIndex {
        endIndex = len(d.GenericDataList)
    }

    d.GenericDataList = d.GenericDataList[startIndex:endIndex]
    return d
}
```

（5）定义podCell类型，实现DataCell接口，用于类型转换

```
//定义podCell类型，实现GetCreateion和GetName方法后，可进行类型转换
type podCell corev1.Pod

func(p podCell) GetCreation() time.Time {
    return p.CreationTimestamp.Time
}

func(p podCell) GetName() string {
    return p.Name
}
```

service/pod.go

（6）定义DataCell到Pod类型转换的方法

断言：判断一个变量是否属于某一种类型，前提是这一类型实现了这一变量类型接口

```
//toCells方法用于将pod类型数组，转换成DataCell类型数组
func(p *pod) toCells(std []corev1.Pod) []DataCell {
    cells := make([]DataCell, len(std))
    for i := range std {
        cells[i] = podCell(std[i])
    }
    return cells
}
//fromCells方法用于将DataCell类型数组，转换成pod类型数组
func(p *pod) fromCells(cells []DataCell) []corev1.Pod {
    pods := make([]corev1.Pod, len(cells))
    for i := range cells {
        //cells[i].(podCell)就使用到了断言,断言后转换成了podCell类型，然后又转换成了Pod类型
        pods[i] = corev1.Pod(cells[i].(podCell))
    }
```

```
    return pods
}
```

2.2 Pod

service/pod.go

（1）定义数据类型

```
package service

import (
    "bytes"
    "context"
    "encoding/json"
    "errors"
    "github.com/wonderivan/logger"
    "io"
    "k8s-demo/config"
    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)
//定义pod类型和Pod对象，用于包外的调用(包是指service目录)，例如Controller
var Pod pod
type pod struct{}

//定义列表的返回内容，Items是pod元素列表，Total为pod元素数量
type PodsResp struct {
    Items []corev1.Pod       `json:"items"`
    Total int                `json:"total"`
}

//定义PodsNp类型，用于返回namespace中pod的数量
type PodsNp struct {
    Namespace  string  `json:"namespace"`
    PodNum     int     `json:"pod_num"`
}
```

（2）获取pod列表

```
//获取pod列表，支持过滤、排序、分页
func(p *pod) GetPods(filterName, namespace string, limit, page int) (podsResp
*PodsResp, err error) {
    //获取podList类型的pod列表
    //context.TODO()用于声明一个空的context上下文，用于List方法内设置这个请求的超时（源码），这里
的常用用法
    //metav1.ListOptions{}用于过滤List数据，如使用label，field等
    //kubectl get services --all-namespaces --field-seletor metadata.namespace !=
default
    podList, err := K8s.ClientSet.CoreV1().Pods(namespace).List(context.TODO(),
metav1.ListOptions{})
    if err != nil {
        //logger用于打印日志
        //return用于返回response内容
        logger.Error(errors.New("获取Pod列表失败, " + err.Error()))
        return nil, errors.New("获取Pod列表失败, " + err.Error())
    }
    //实例化dataSelector对象
```

```
    selectableData := &dataSelector{
        GenericDataList: p.toCells(podList.Items),
        dataSelectQuery:    &DataSelectQuery{
            FilterQuery:   &FilterQuery{Name: filterName},
            PaginateQuery: &PaginateQuery{
                Limit: limit,
                Page:  page,
            },
        },
    }
    //先过滤
    filtered := selectableData.Filter()
    total := len(filtered.GenericDataList)
    //再排序和分页
    data := filtered.Sort().Paginate()

    //将[]DataCell类型的pod列表转为v1.pod列表
    pods := p.fromCells(data.GenericDataList)

    return &PodsResp{
        Items: pods,
        Total: total,
    }, nil
}
```

(3) 获取Pod详情

```
//获取pod详情
func(p *pod) GetPodDetail(podName, namespace string) (pod *corev1.Pod, err error) {
    pod, err = K8s.ClientSet.CoreV1().Pods(namespace).Get(context.TODO(), podName,
metav1.GetOptions{})
    if err != nil {
        logger.Error(errors.New("获取Pod详情失败, " + err.Error()))
        return nil, errors.New("获取Pod详情失败, " + err.Error())
    }

    return pod, nil
}
```

(4) 删除Pod

```
//删除pod
func(p *pod) DeletePod(podName, namespace string) (err error) {
    err = K8s.ClientSet.CoreV1().Pods(namespace).Delete(context.TODO(), podName,
metav1.DeleteOptions{})
    if err != nil {
        logger.Error(errors.New("删除pod失败, " + err.Error()))
        return errors.New("删除pod失败, " + err.Error())
    }

    return nil
}
```

(5) 更新Pod

```
//更新pod
//content参数是请求中传入的pod对象的json数据
func(p *pod) UpdatePod(podName, namespace, content string) (err error) {
```

```
    var pod = &corev1.Pod{}
    //反序列化为pod对象
    err = json.Unmarshal([]byte(content), pod)
    if err != nil {
        logger.Error(errors.New("反序列化失败, " + err.Error()))
        return errors.New("反序列化失败, " + err.Error())
    }
    //更新pod
    _, err = K8s.ClientSet.CoreV1().Pods(namespace).Update(context.TODO(), pod,
metav1.UpdateOptions{})
    if err != nil {
        logger.Error(errors.New("更新Pod失败, " + err.Error()))
        return errors.New("更新Pod失败, " + err.Error())
    }
    return nil
}
```

(6) 获取Pod中的容器名

```
//获取pod容器
func(p *pod) GetPodContainer(podName, namespace string) (containers []string, err
error) {
    //获取pod详情
    pod, err := p.GetPodDetail(podName, namespace)
    if err != nil {
        return nil, err
    }
    //从pod对象中拿到容器名
    for _, container := range pod.Spec.Containers {
        containers = append(containers, container.Name)
    }

    return containers, nil
}
```

(7) 获取容器日志

```
//获取pod内容器日志
func(p *pod) GetPodLog(containerName, podName, namespace string) (log string, err
error) {
    //设置日志的配置，容器名、tail的行数
    lineLimit := int64(config.PodLogTailLine)
    option := &corev1.PodLogOptions{
        Container: containerName,
        TailLines: &lineLimit,
    }
    //获取request实例
    req := K8s.ClientSet.CoreV1().Pods(namespace).GetLogs(podName, option)
    //发起request请求，返回一个io.ReadCloser类型（等同于response.body）
    podLogs, err := req.Stream(context.TODO())
    if err != nil {
        logger.Error(errors.New("获取PodLog失败, " + err.Error()))
        return "", errors.New("获取PodLog失败, " + err.Error())
    }
    defer podLogs.Close()
    //将response body写入到缓冲区，目的是为了转成string返回
    buf := new(bytes.Buffer)
    _, err = io.Copy(buf, podLogs)
```

```
    if err != nil {
        logger.Error(errors.New("复制PodLog失败, " + err.Error()))
        return "", errors.New("复制PodLog失败, " + err.Error())
    }

    return buf.String(), nil
}
```

(8) 获取每个namespace的pod数量

```
//获取每个namespace的pod数量
func(p *pod) GetPodNumPerNp() (podsNps []*PodsNp, err error) {
    //获取namespace列表
    namespaceList, err := K8s.ClientSet.CoreV1().Namespaces().List(context.TODO(),
metav1.ListOptions{})
    if err != nil {
        return nil, err
    }
    for _, namespace := range namespaceList.Items {
        //获取pod列表
        podList, err :=
K8s.ClientSet.CoreV1().Pods(namespace.Name).List(context.TODO(), metav1.ListOptions{})
        if err != nil {
            return nil, err
        }
        //组装数据
        podsNp := &PodsNp{
            Namespace: namespace.Name,
            PodNum:    len(podList.Items),
        }
        //添加到podsNps数组中
        podsNps = append(podsNps, podsNp)
    }
    return podsNps, nil
}
```

controller/pod.go

(9) 编写Controller层的pod代码

```
package controller

import (
    "github.com/gin-gonic/gin"
    "github.com/wonderivan/logger"
    "k8s-demo/service"
    "net/http"
)

var Pod pod

type pod struct {}

//Controller中的方法入参是gin.Context，用于从上下文中获取请求参数及定义响应内容
//流程：绑定参数->调用service代码->根据调用结果响应具体内容

//获取pod列表，支持过滤、排序、分页
func(p *pod) GetPods(ctx *gin.Context) {
```

```
    //匿名结构体，用于声明入参，get请求为form格式，其他请求为json格式
    params := new(struct {
        FilterName  string `form:"filter_name"`
        Namespace   string `form:"namespace"`
        Page        int    `form:"page"`
        Limit       int    `form:"limit"`
    })
    //绑定参数，给匿名结构体中的属性赋值，值是入参
    //form格式使用ctx.Bind方法，json格式使用ctx.ShouldBindJSON方法
    if err := ctx.Bind(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        //ctx.JSON方法用于返回响应内容，入参是状态码和响应内容，响应内容放入gin.H的map中
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    //service中的的方法通过 包名.结构体变量名.方法名 使用，serivce.Pod.GetPods()
    data, err := service.Pod.GetPods(params.FilterName, params.Namespace,
params.Limit, params.Page)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    ctx.JSON(http.StatusOK, gin.H{
        "msg": "获取Pod列表成功",
        "data": data,
    })
}
//获取pod详情
func(p *pod) GetPodDetail(ctx *gin.Context) {
    params := new(struct{
        PodName   string `form:"pod_name"`
        Namespace string `form:"namespace"`
    })
    if err := ctx.Bind(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    data, err := service.Pod.GetPodDetail(params.PodName, params.Namespace)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
        "msg": "获取Pod详情成功",
```

```
        "data": data,
    })
}

//删除pod
func(p *pod) DeletePod(ctx *gin.Context) {
    params := new(struct{
        PodName   string `json:"pod_name"`
        Namespace string `json:"namespace"`
    })
    //PUT请求，绑定参数方法改为ctx.ShouldBindJSON
    if err := ctx.ShouldBindJSON(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    err := service.Pod.DeletePod(params.PodName, params.Namespace)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
        "msg": "删除Pod成功",
        "data": nil,
    })
}

//更新pod
func(p *pod) UpdatePod(ctx *gin.Context) {
    params := new(struct{
        PodName   string `json:"pod_name"`
        Namespace string `json:"namespace"`
        Content   string `json:"content"`
    })
    //PUT请求，绑定参数方法改为ctx.ShouldBindJSON
    if err := ctx.ShouldBindJSON(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    err := service.Pod.UpdatePod(params.PodName, params.Namespace, params.Content)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
```

```
        "msg": "更新Pod成功",
        "data": nil,
    })
}

//获取pod容器
func(p *pod) GetPodContainer(ctx *gin.Context) {
    params := new(struct{
        PodName   string `form:"pod_name"`
        Namespace string `form:"namespace"`
    })
    //GET请求，绑定参数方法改为ctx.Bind
    if err := ctx.Bind(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    data, err := service.Pod.GetPodContainer(params.PodName, params.Namespace)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
        "msg": "获取Pod容器成功",
        "data": data,
    })
}

//获取pod中容器日志
func(p *pod) GetPodLog(ctx *gin.Context) {
    params := new(struct{
        ContainerName string `form:"container_name"`
        PodName       string `form:"pod_name"`
        Namespace     string `form:"namespace"`
    })
    //GET请求，绑定参数方法改为ctx.Bind
    if err := ctx.Bind(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    data, err := service.Pod.GetPodLog(params.ContainerName, params.PodName,
params.Namespace)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
```

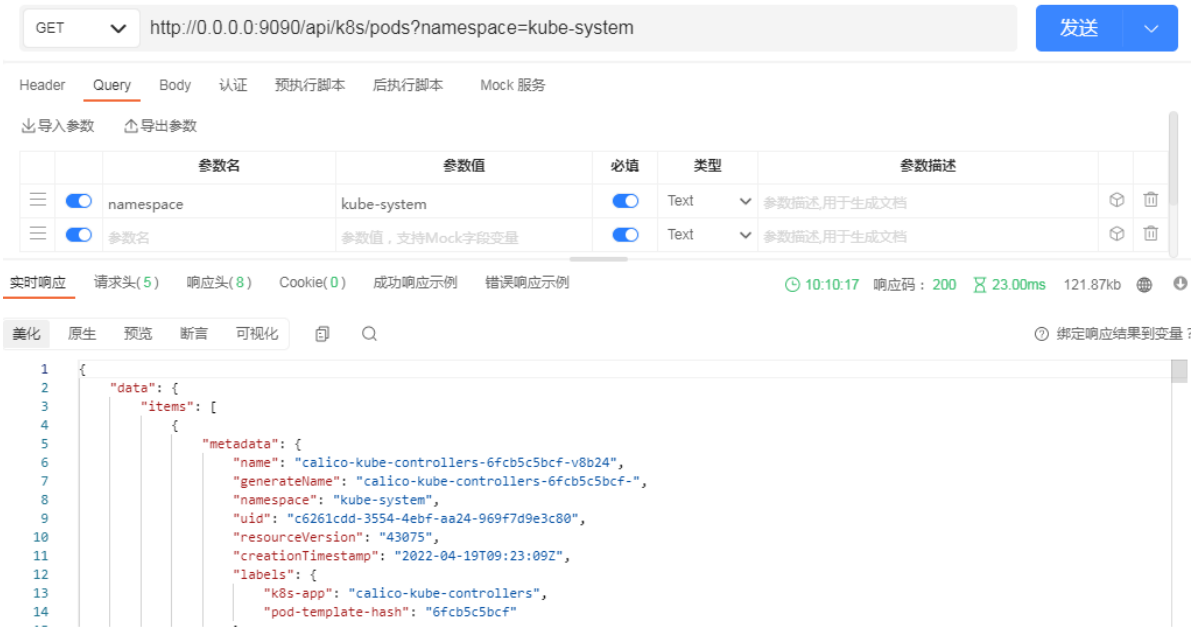
```
    }
    ctx.JSON(http.StatusOK, gin.H{
        "msg": "获取Pod中容器日志成功",
        "data": data,
    })
}

//获取每个namespace的pod数量
func(p *pod) GetPodNumPerNp(ctx *gin.Context) {
    data, err := service.Pod.GetPodNumPerNp()
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    ctx.JSON(http.StatusOK, gin.H{
        "msg": "获取每个namespace的pod数量成功",
        "data": data,
    })
}
```

controller/router.go

（10）定义路由规则

```
package controller

import (
    "github.com/gin-gonic/gin"
)

var Router router

type router struct{}

func(r *router) InitApiRouter(router *gin.Engine) {
    router.
        //pod操作
        GET("/api/k8s/pods", Pod.GetPods).
        GET("/api/k8s/pod/detail", Pod.GetPodDetail).
        DELETE("/api/k8s/pod/del", Pod.DeletePod).
        PUT("/api/k8s/pod/update", Pod.UpdatePod).
        GET("/api/k8s/pod/container", Pod.GetPodContainer).
        GET("/api/k8s/pod/log", Pod.GetPodLog).
        GET("/api/k8s/pod/numnp", Pod.GetPodNumPerNp)
}
```

（11）使用apihost做接口测试



2.2 Deployment

service/dataselector.go

实现DataCell接口

```
type deploymentCell appsv1.Deployment

func(d deploymentCell) GetCreation() time.Time {
    return d.CreationTimestamp.Time
}

func(d deploymentCell) GetName() string {
    return d.Name
}
```

service/deployment.go

开发deployment功能，包括：

(1) 列表

(2) 获取Deployment详情

(3) 修改Deployment副本数

(4) 创建Deployment

(5) 删除Deployment

(6) 重启Deployment

(7) 更新Deployment

(8) 获取每个namespace的Deployment数量

```
package service

import (
    "context"
    "encoding/json"
    "errors"
    "github.com/wonderivan/logger"
```

```
    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/api/resource"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/util/intstr"
    "strconv"
    "time"
)

var Deployment deployment

type deployment struct{}

//定义列表的返回内容，Items是deployment元素列表，Total为deployment元素数量
type DeploymentsResp struct {
    Items []appsv1.Deployment `json:"items"`
    Total int                 `json:"total"`
}
//定义DeployCreate结构体，用于创建deployment需要的参数属性的定义
type DeployCreate struct {
    Name           string  `json:"name"`
    Namespace      string  `json:"namespace"`
    Replicas       int32   `json:"replicas"`
    Image          string  `json:"image"`
    Label          map[string]string  `json:"label"`
    Cpu            string  `json:"cpu"`
    Memory         string  `json:"memory"`
    ContainerPort  int32   `json:"container_port"`
    HealthCheck    bool    `json:"health_check"`
    HealthPath     string  `json:"health_path"`
}
//定义DeploysNp类型，用于返回namespace中deployment的数量
type DeploysNp struct {
    Namespace  string  `json:"namespace"`
    DeployNum  int     `json:"deployment_num"`
}

//获取deployment列表，支持过滤、排序、分页
func(d *deployment) GetDeployments(filterName, namespace string, limit, page int)
(deploymentsResp *DeploymentsResp, err error) {
    //获取deploymentList类型的deployment列表
    deploymentList, err :=
K8s.ClientSet.AppsV1().Deployments(namespace).List(context.TODO(),
metav1.ListOptions{})
    if err != nil {
        logger.Error(errors.New("获取Deployment列表失败, " + err.Error()))
        return nil, errors.New("获取Deployment列表失败, " + err.Error())
    }
    //将deploymentList中的deployment列表(Items)，放进dataselector对象中，进行排序
    selectableData := &dataSelector{
        GenericDataList: d.toCells(deploymentList.Items),
        dataSelectQuery:    &DataSelectQuery{
            FilterQuery:   &FilterQuery{Name: filterName},
            PaginateQuery: &PaginateQuery{
                Limit: limit,
                Page:  page,
            },
        },
```

```
    }

    filtered := selectableData.Filter()
    total := len(filtered.GenericDataList)
    data := filtered.Sort().Paginate()

    //将[]DataCell类型的deployment列表转为appsv1.deployment列表
    deployments := d.fromCells(data.GenericDataList)

    return &DeploymentsResp{
        Items: deployments,
        Total: total,
    }, nil
}

//获取deployment详情
func(d *deployment) GetDeploymentDetail(deploymentName, namespace string) (deployment
*appsv1.Deployment, err error) {
    deployment, err =
K8s.ClientSet.AppsV1().Deployments(namespace).Get(context.TODO(), deploymentName,
metav1.GetOptions{})
    if err != nil {
        logger.Error(errors.New("获取Deployment详情失败, " + err.Error()))
        return nil, errors.New("获取Deployment详情失败, " + err.Error())
    }

    return deployment, nil
}

//设置deployment副本数
func(d *deployment) ScaleDeployment(deploymentName, namespace string, scaleNum int)
(replica int32, err error) {
    //获取autoscalingv1.Scale类型的对象，能点出当前的副本数
    scale, err :=
K8s.ClientSet.AppsV1().Deployments(namespace).GetScale(context.TODO(), deploymentName,
metav1.GetOptions{})
    if err != nil {
        logger.Error(errors.New("获取Deployment副本数信息失败, " + err.Error()))
        return 0, errors.New("获取Deployment副本数信息失败, " + err.Error())
    }
    //修改副本数
    scale.Spec.Replicas = int32(scaleNum)
    //更新副本数，传入scale对象
    newScale, err :=
K8s.ClientSet.AppsV1().Deployments(namespace).UpdateScale(context.TODO(),
deploymentName, scale, metav1.UpdateOptions{})
    if err != nil {
        logger.Error(errors.New("更新Deployment副本数信息失败, " + err.Error()))
        return 0, errors.New("更新Deployment副本数信息失败, " + err.Error())
    }

    return newScale.Spec.Replicas, nil
}

//创建deployment,接收DeployCreate对象
func(d *deployment) CreateDeployment(data *DeployCreate) (err error) {
    //将data中的数据组装成appsv1.Deployment对象
    deployment := &appsv1.Deployment{
```

```
        //ObjectMeta中定义资源名、命名空间以及标签
        ObjectMeta: metav1.ObjectMeta{
            Name: data.Name,
            Namespace: data.Namespace,
            Labels: data.Label,
        },
        //Spec中定义副本数、选择器、以及pod属性
        Spec: appsv1.DeploymentSpec{
            Replicas: &data.Replicas,
            Selector: &metav1.LabelSelector{
                MatchLabels: data.Label,
            },
            Template: corev1.PodTemplateSpec{
                //定义pod名和标签
                ObjectMeta: metav1.ObjectMeta{
                    Name: data.Name,
                    Labels: data.Label,
                },
                //定义容器名、镜像和端口
                Spec: corev1.PodSpec{
                    Containers: []corev1.Container{
                        {
                            Name: data.Name,
                            Image: data.Image,
                            Ports: []corev1.ContainerPort{
                                {
                                    Name: "http",
                                    Protocol: corev1.ProtocolTCP,
                                    ContainerPort: 80,
                                },
                            },
                        },
                    },
                },
            },
        },
        //Status定义资源的运行状态，这里由于是新建，传入空的appsv1.DeploymentStatus{}对象即可
        Status: appsv1.DeploymentStatus{},
    }
    //判断是否打开健康检查功能，若打开，则定义ReadinessProbe和LivenessProbe
    if data.HealthCheck {
        //设置第一个容器的ReadinessProbe，因为我们pod中只有一个容器，所以直接使用index 0即可
        //若pod中有多个容器，则这里需要使用for循环去定义了
        deployment.Spec.Template.Spec.Containers[0].ReadinessProbe = &corev1.Probe{
            Handler: corev1.Handler{
                HTTPGet: &corev1.HTTPGetAction{
                    Path: data.HealthPath,
                    //intstr.IntOrString的作用是端口可以定义为整型，也可以定义为字符串
                    //Type=0则表示表示该结构体实例内的数据为整型，转json时只使用IntVal的数据
                    //Type=1则表示表示该结构体实例内的数据为字符串，转json时只使用StrVal的数据
                    Port: intstr.IntOrString{
                        Type: 0,
                        IntVal: data.ContainerPort,
                    },
                },
            },
            //初始化等待时间
            InitialDelaySeconds: 5,
```

```
            //超时时间
            TimeoutSeconds:      5,
            //执行间隔
            PeriodSeconds:       5,
        }
        deployment.Spec.Template.Spec.Containers[0].LivenessProbe = &corev1.Probe{
            Handler: corev1.Handler{
                HTTPGet: &corev1.HTTPGetAction{
                    Path: data.HealthPath,
                    Port: intstr.IntOrString{
                        Type: 0,
                        IntVal: data.ContainerPort,
                    },
                },
            },
            InitialDelaySeconds: 15,
            TimeoutSeconds:      5,
            PeriodSeconds:       5,
        }
        //定义容器的limit和request资源
        deployment.Spec.Template.Spec.Containers[0].Resources.Limits =
map[corev1.ResourceName]resource.Quantity{
            corev1.ResourceCPU : resource.MustParse(data.Cpu),
            corev1.ResourceMemory : resource.MustParse(data.Memory),
        }
        deployment.Spec.Template.Spec.Containers[0].Resources.Requests =
map[corev1.ResourceName]resource.Quantity{
            corev1.ResourceCPU : resource.MustParse(data.Cpu),
            corev1.ResourceMemory : resource.MustParse(data.Memory),
        }
    }
    //调用sdk创建deployment
    _, err = K8s.ClientSet.AppsV1().Deployments(data.Namespace).Create(context.TODO(),
deployment, metav1.CreateOptions{})
    if err != nil {
        logger.Error(errors.New("创建Deployment失败, " + err.Error()))
        return errors.New("创建Deployment失败, " + err.Error())
    }

    return nil
}

//删除deployment
func(d *deployment) DeleteDeployment(deploymentName, namespace string) (err error) {
    err = K8s.ClientSet.AppsV1().Deployments(namespace).Delete(context.TODO(),
deploymentName, metav1.DeleteOptions{})
    if err != nil {
        logger.Error(errors.New("删除Deployment失败, " + err.Error()))
        return errors.New("删除Deployment失败, " + err.Error())
    }

    return nil
}

//重启deployment
func(d *deployment) RestartDeployment(deploymentName, namespace string) (err error) {
    //此功能等同于一下kubectl命令
    //kubectl deployment ${service} -p \
```

```
    //'{"spec":{"template":{"spec":{"containers":[{"name":"'"${service}"'","env":
[{"name":"RESTART_","value":"'$(date +%s)'"}]}]}}}}'

    //使用patchData Map组装数据
    patchData := map[string]interface{}{
        "spec": map[string]interface{}{
            "template": map[string]interface{}{
                "spec": map[string]interface{}{
                    "containers": []map[string]interface{}{
                        {"name": deploymentName,
                            "env": []map[string]string{{
                                "name":  "RESTART_",
                                "value": strconv.FormatInt(time.Now().Unix(), 10),
                            }},
                        },
                    },
                },
            },
        },
    }
    //序列化为字节，因为patch方法只接收字节类型参数
    patchByte, err := json.Marshal(patchData)
    if err != nil {
        logger.Error(errors.New("json序列化失败, " + err.Error()))
        return errors.New("json序列化失败, " + err.Error())
    }
    //调用patch方法更新deployment
    _, err = K8s.ClientSet.AppsV1().Deployments(namespace).Patch(context.TODO(),
deploymentName, "application/strategic-merge-patch+json", patchByte,
metav1.PatchOptions{})
    if err != nil {
        logger.Error(errors.New("重启Deployment失败, " + err.Error()))
        return errors.New("重启Deployment失败, " + err.Error())
    }

    return nil
}

//更新deployment
func(d *deployment) UpdateDeployment(namespace, content string) (err error) {
    var deploy = &appsv1.Deployment{}

    err = json.Unmarshal([]byte(content), deploy)
    if err != nil {
        logger.Error(errors.New("反序列化失败, " + err.Error()))
        return errors.New("反序列化失败, " + err.Error())
    }

    _, err = K8s.ClientSet.AppsV1().Deployments(namespace).Update(context.TODO(),
deploy, metav1.UpdateOptions{})
    if err != nil {
        logger.Error(errors.New("更新Deployment失败, " + err.Error()))
        return errors.New("更新Deployment失败, " + err.Error())
    }
    return nil
}

//获取每个namespace的deployment数量
```

```
func(d *deployment) GetDeployNumPerNp() (deploysNps []*DeploysNp, err error) {
    namespaceList, err := K8s.ClientSet.CoreV1().Namespaces().List(context.TODO(),
metav1.ListOptions{})
    if err != nil {
        return nil, err
    }
    for _, namespace := range namespaceList.Items {
        deploymentList, err :=
K8s.ClientSet.AppsV1().Deployments(namespace.Name).List(context.TODO(),
metav1.ListOptions{})
        if err != nil {
            return nil, err
        }

        deploysNp := &DeploysNp{
            Namespace: namespace.Name,
            DeployNum:    len(deploymentList.Items),
        }

        deploysNps = append(deploysNps, deploysNp)
    }
    return deploysNps, nil
}

func(d *deployment) toCells(std []appsv1.Deployment) []DataCell {
    cells := make([]DataCell, len(std))
    for i := range std {
        cells[i] = deploymentCell(std[i])
    }
    return cells
}

func(d *deployment) fromCells(cells []DataCell) []appsv1.Deployment {
    deployments := make([]appsv1.Deployment, len(cells))
    for i := range cells {
        deployments[i] = appsv1.Deployment(cells[i].(deploymentCell))
    }

    return deployments
}
```

controller/deployment.go

```
package controller

import (
    "fmt"
    "github.com/gin-gonic/gin"
    "github.com/wonderivan/logger"
    "k8s-demo/service"
    "net/http"
)

var Deployment deployment

type deployment struct {}

//获取deployment列表，支持过滤、排序、分页
```

```
func(d *deployment) GetDeployments(ctx *gin.Context) {
    params := new(struct {
        FilterName  string `form:"filter_name"`
        Namespace   string `form:"namespace"`
        Page        int    `form:"page"`
        Limit       int    `form:"limit"`
    })
    if err := ctx.Bind(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    data, err := service.Deployment.GetDeployments(params.FilterName,
params.Namespace, params.Limit, params.Page)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    ctx.JSON(http.StatusOK, gin.H{
        "msg": "获取Deployment列表成功",
        "data": data,
    })
}
//获取deployment详情
func(d *deployment) GetDeploymentDetail(ctx *gin.Context) {
    params := new(struct{
        DeploymentName   string `form:"deployment_name"`
        Namespace        string `form:"namespace"`
    })
    if err := ctx.Bind(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    data, err := service.Deployment.GetDeploymentDetail(params.DeploymentName,
params.Namespace)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
        "msg": "获取Deployment详情成功",
        "data": data,
    })
```

```
}

//创建deployment
func(d *deployment) CreateDeployment(ctx *gin.Context) {
    var (
        deployCreate = new(service.DeployCreate)
        err error
    )

    if err = ctx.ShouldBindJSON(deployCreate); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    if err = service.Deployment.CreateDeployment(deployCreate); err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
    }

    ctx.JSON(http.StatusOK, gin.H{
        "msg": "创建Deployment成功",
        "data": nil,
    })
}

//设置deployment副本数
func(d *deployment) ScaleDeployment(ctx * gin.Context) {
    params := new(struct{
        DeploymentName  string  `json:"deployment_name"`
        Namespace       string  `json:"namespace"`
        ScaleNum        int     `json:"scale_num"`
    })
    //PUT请求，绑定参数方法改为ctx.ShouldBindJSON
    if err := ctx.ShouldBindJSON(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    data, err := service.Deployment.ScaleDeployment(params.DeploymentName,
params.Namespace, params.ScaleNum)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
```

```
        "msg": "设置Deployment副本数成功",
        "data": fmt.Sprintf("最新副本数: %d", data),
    })
}

//删除deployment
func(d *deployment) DeleteDeployment(ctx *gin.Context) {
    params := new(struct{
        DeploymentName  string  `json:"deployment_name"`
        Namespace       string  `json:"namespace"`
    })
    //DELETE请求，绑定参数方法改为ctx.ShouldBindJSON
    if err := ctx.ShouldBindJSON(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    err := service.Deployment.DeleteDeployment(params.DeploymentName,
params.Namespace)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
        "msg": "删除Deployment成功",
        "data": nil,
    })
}

//重启deployment
func(d *deployment) RestartDeployment(ctx *gin.Context) {
    params := new(struct{
        DeploymentName  string  `json:"deployment_name"`
        Namespace       string  `json:"namespace"`
    })
    //PUT请求，绑定参数方法改为ctx.ShouldBindJSON
    if err := ctx.ShouldBindJSON(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    err := service.Deployment.RestartDeployment(params.DeploymentName,
params.Namespace)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
```

```
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
        "msg": "重启Deployment成功",
        "data": nil,
    })
}

//更新deployment
func(d *deployment) UpdateDeployment(ctx *gin.Context) {
    params := new(struct{
        Namespace       string  `json:"namespace"`
        Content         string  `json:"content"`
    })
    //PUT请求，绑定参数方法改为ctx.ShouldBindJSON
    if err := ctx.ShouldBindJSON(params); err != nil {
        logger.Error("Bind请求参数失败, " + err.Error())
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    err := service.Deployment.UpdateDeployment(params.Namespace, params.Content)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }
    ctx.JSON(http.StatusOK, gin.H{
        "msg": "更新Deployment成功",
        "data": nil,
    })
}

//获取每个namespace的pod数量
func(d *deployment) GetDeployNumPerNp(ctx *gin.Context) {
    data, err := service.Deployment.GetDeployNumPerNp()
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{
            "msg": err.Error(),
            "data": nil,
        })
        return
    }

    ctx.JSON(http.StatusOK, gin.H{
        "msg": "获取每个namespace的deployment数量成功",
        "data": data,
    })
}
```

(9) 定义路由规则

```
package controller

import (
    "github.com/gin-gonic/gin"
)

var Router router

type router struct{}

func(r *router) InitApiRouter(router *gin.Engine) {
    router.
        //登录
        POST("/api/login", Login.Auth).
        //工作流
        POST("/api/k8s/workflow/create", Workflow.Create).
        //pod操作
        GET("/api/k8s/pods", Pod.GetPods).
        GET("/api/k8s/pod/detail", Pod.GetPodDetail).
        DELETE("/api/k8s/pod/del", Pod.DeletePod).
        PUT("/api/k8s/pod/update", Pod.UpdatePod).
        GET("/api/k8s/pod/container", Pod.GetPodContainer).
        GET("/api/k8s/pod/log", Pod.GetPodLog).
        GET("/api/k8s/pod/numnp", Pod.GetPodNumPerNp).
        //deployment操作
        GET("/api/k8s/deployments", Deployment.GetDeployments).
        GET("/api/k8s/deployment/detail", Deployment.GetDeploymentDetail).
        PUT("/api/k8s/deployment/scale", Deployment.ScaleDeployment).
        DELETE("/api/k8s/deployment/del", Deployment.DeleteDeployment).
        PUT("/api/k8s/deployment/restart", Deployment.RestartDeployment).
        PUT("/api/k8s/deployment/update", Deployment.UpdateDeployment).
        GET("/api/k8s/deployment/numnp", Deployment.GetDeployNumPerNp).
        POST("/api/k8s/deployment/create", Deployment.CreateDeployment)
}
```

2.3 DaemonSet

> 阿良教育： www.aliangedu.cn

注意：以下功能与Pod和Deployment是实现方式一致，代码就不贴了

（1）列表

（2）获取DaemonSet详情

（3）删除DaemonSet

（4）更新DaemonSet

2.4 StatefulSet

（1）列表

（2）获取StatefulSet详情

（3）删除StatefulSet

（4）更新StatefulSet

3、集群

注意：以下3个资源是集群维度的，没有Namespace的概念

3.1 Node

（1）列表

（2）获取Node详情

3.2 Namespace

（1）列表

（2）获取Namespace详情

（3）删除Namespace

3.3 PersistentVolume

（1）列表

（2）获取Pv详情

（3）删除Pv

4、 负载均衡

4.1 Service

注意：代码只贴上面未实现过的功能，蓝色部分功能

（1）列表

（2）获取Service详情

（3）创建Service

（4）删除Service

（5）更新Service

创建Service

```
//定义ServiceCreate结构体，用于创建service需要的参数属性的定义
type ServiceCreate struct {
    Name           string  `json:"name"`
    Namespace      string  `json:"namespace"`
    Type           string  `json:"type"`
    ContainerPort  int32   `json:"container_port"`
    Port           int32   `json:"port"`
    NodePort       int32   `json:"node_port"`
    Label          map[string]string  `json:"label"`
}

//创建service,,接收ServiceCreate对象
func(s *servicev1) CreateService(data *ServiceCreate) (err error) {
    //将data中的数据组装成corev1.Service对象
    service := &corev1.Service{
        //ObjectMeta中定义资源名、命名空间以及标签
        ObjectMeta: metav1.ObjectMeta{
            Name: data.Name,
```

```
            Namespace: data.Namespace,
            Labels: data.Label,
        },
        //Spec中定义类型，端口，选择器
        Spec: corev1.ServiceSpec{
            Type: corev1.ServiceType(data.Type),
            Ports: []corev1.ServicePort{
                {
                    Name: "http",
                    Port: data.Port,
                    Protocol: "TCP",
                    TargetPort: intstr.IntOrString{
                        Type: 0,
                        IntVal: data.ContainerPort,
                    },
                },
            },
            Selector: data.Label,
        },
    }
    //默认ClusterIP,这里是判断NodePort,添加配置
    if data.NodePort != 0 && data.Type == "NodePort" {
        service.Spec.Ports[0].NodePort = data.NodePort
    }
    //创建Service
    _, err = K8s.ClientSet.CoreV1().Services(data.Namespace).Create(context.TODO(),
service, metav1.CreateOptions{})
    if err != nil {
        logger.Error(errors.New("创建Service失败, " + err.Error()))
        return errors.New("创建Service失败, " + err.Error())
    }

    return nil
}
```

对标yaml

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
  namespace: default
spec:
  selector:
    app: myapp
  ports:
  - name: http
    port: 80 #service的端口
    protocol:  tcp  #协议
    targetPort: 80  #pod的端口
```

4.2 Ingress

(1) 列表

(2) 获取Ingress详情

(3) 创建Ingress

(4) 删除Ingress

(5) 更新Ingress

创建Ingress

```
//定义ServiceCreate结构体，用于创建service需要的参数属性的定义
type IngressCreate struct {
    Name         string  `json:"name"`
    Namespace    string  `json:"namespace"`
    Label        map[string]string  `json:"label"`
    Hosts        map[string][]*HttpPath `json:"hosts"`
}
//定义ingress的path结构体
type HttpPath struct {
    Path         string         `json:"path"`
    PathType     nwv1.PathType  `json:"path_type"`
    ServiceName  string         `json:"service_name"`
    ServicePort  int32          `json:"service_port"`
}

//创建ingress
func(i *ingress) CreateIngress(data *IngressCreate) (err error) {
    //声明nwv1.IngressRule和nwv1.HTTPIngressPath变量，后面组装数据于鏊用到
    var ingressRules []nwv1.IngressRule
    var httpIngressPATHs []nwv1.HTTPIngressPath
    //将data中的数据组装成nwv1.Ingress对象
    ingress := &nwv1.Ingress{
        ObjectMeta: metav1.ObjectMeta{
            Name: data.Name,
            Namespace: data.Namespace,
            Labels: data.Label,
        },
        Status: nwv1.IngressStatus{},
    }
    //第一层for循环是将host组装成nwv1.IngressRule类型的对象
    // 一个host对应一个ingressrule，每个ingressrule中包含一个host和多个path
    for key, value := range data.Hosts {
        ir := nwv1.IngressRule{
            Host: key,
            //这里现将nwv1.HTTPIngressRuleValue类型中的Paths置为空，后面组装好数据再赋值
            IngressRuleValue: nwv1.IngressRuleValue{
                HTTP: &nwv1.HTTPIngressRuleValue{Paths:nil},
            },
        }
        //第二层for循环是将path组装成nwv1.HTTPIngressPath类型的对象
        for _, httpPath := range value {
            hip := nwv1.HTTPIngressPath{
                Path: httpPath.Path,
                PathType: &httpPath.PathType,
                Backend: nwv1.IngressBackend{
```

```
                    Service: &nwv1.IngressServiceBackend{
                        Name: httpPath.ServiceName,
                        Port: nwv1.ServiceBackendPort{
                            Number: httpPath.ServicePort,
                        },
                    },
                },
            }
            //将每个hip对象组装成数组
            httpIngressPATHs = append(httpIngressPATHs, hip)
        }
        //给Paths赋值，前面置为空了
        ir.IngressRuleValue.HTTP.Paths = httpIngressPATHs
        //将每个ir对象组装成数组，这个ir对象就是IngressRule，每个元素是一个host和多个path
        ingressRules = append(ingressRules, ir)
    }
    //将ingressRules对象加入到ingress的规则中
    ingress.Spec.Rules = ingressRules
    //创建ingress
    _, err =
K8s.ClientSet.NetworkingV1().Ingresses(data.Namespace).Create(context.TODO(), ingress,
metav1.CreateOptions{})
    if err != nil {
        logger.Error(errors.New("创建Ingress失败, " + err.Error()))
        return errors.New("创建Ingress失败, " + err.Error())
    }

    return nil
}
```

对标yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-myapp
  namespace: default
spec:
  rules:
  - host: www.xxx.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: myapp-svc
            port:
              number: 80
```

5、存储与配置

5.1 ConfigMap

（1）列表

（2）获取Configmap详情

（3）删除Configmap

（4）更新Configmap

5.2 Secret

（1）列表

（2）获取Secret详情

（3）删除Secret

（4）更新Secret

5.3 PersistentVolumeClaim

（1）列表

（2）获取Pvc详情

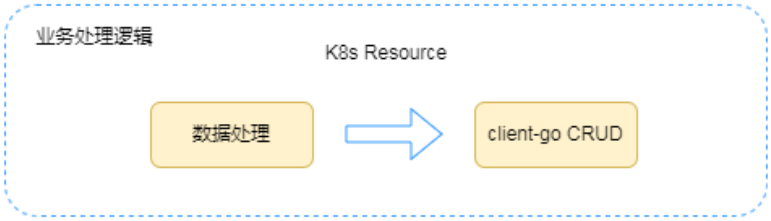
（3）删除Pvc

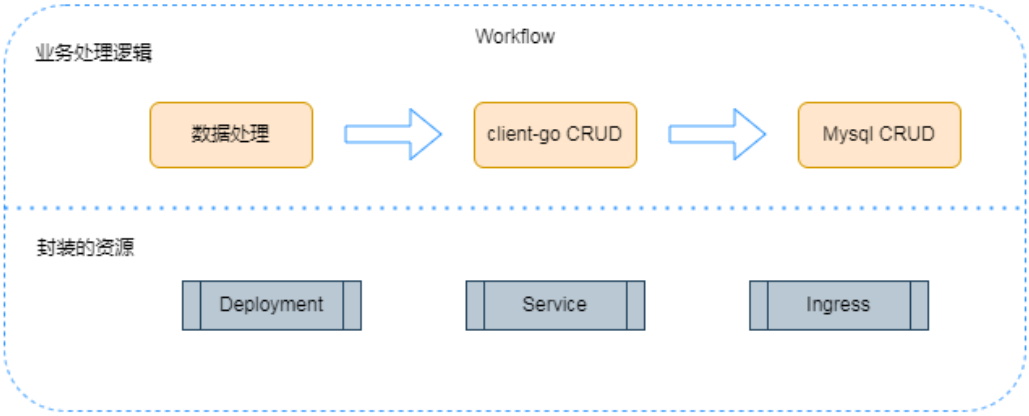
（4）更新Pvc

6、工作流

> 阿良教育： www.aliangedu.cn

6.1 流程设计





6.2 数据库操作（GORM）

（1）初始化数据库

db/db.go

```
package db

import (
    "fmt"
    "github.com/jinzhu/gorm" //gorm库
    _ "github.com/jinzhu/gorm/dialects/mysql" //gorm对应的mysql驱动
    "github.com/wonderivan/logger"
    "k8s-demo/config"
    "time"
)

var (
    isInit bool
    GORM   *gorm.DB
    err    error
)

//db的初始化函数，与数据库建立连接
func Init() {
    //判断是否已经初始化了
    if isInit {
        return
    }
    //组装连接配置
    //parseTime是查询结果是否自动解析为时间
    //loc是Mysql的时区设置
    dsn := fmt.Sprintf("%s:%s@tcp(%s:%d)/%s?charset=utf8&parseTime=True&loc=Local",
```

```
            config.DbUser,
            config.DbPwd,
            config.DbHost,
            config.DbPort,
            config.DbName)
    //与数据库建立连接，生成一个*gorm.DB类型的对象
    GORM, err = gorm.Open(config.DbType, dsn)
    if err != nil {
        panic("数据库连接失败" + err.Error())
    }

    //打印sql语句
    //GORM.LogMode(true)

    //开启连接池
    // 连接池最大允许的空闲连接数，如果没有sql任务需要执行的连接数大于20，超过的连接会被连接池关闭
    GORM.DB().SetMaxIdleConns(config.MaxIdleConns)
    // 设置了连接可复用的最大时间
    GORM.DB().SetMaxOpenConns(config.MaxOpenConns)
    // 设置了连接可复用的最大时间
    GORM.DB().SetConnMaxLifetime(time.Duration(config.MaxLifeTime))

    isInit = true
    logger.Info("连接数据库成功!")
}

//db的关闭函数
func Close() error {
    return GORM.Close()
}
```

SetMaxOpenConns

默认情况下，连接池的最大数量是没有限制的。一般来说，连接数越多，访问数据库的性能越高。但是系统资源不是无限的，数据库的并发能力也不是无限的。因此为了减少系统和数据库崩溃的风险，可以给并发连接数设置一个上限，这个数值一般不超过进程的最大文件句柄打开数，不超过数据库服务自身支持的并发连接数，比如1000。

SetMaxIdleConns

理论上maxIdleConns连接的上限越高，也即允许在连接池中的空闲连接最大值越大，可以有效减少连接创建和销毁的次数，提高程序的性能。但是连接对象也是占用内存资源的，而且如果空闲连接越多，存在于连接池内的时间可能越长。`连接在经过一段时间后有可能会变得不可用`，而这时连接还在连接池内没有回收的话，后续被征用的时候就会出问题。一般建议maxIdleConns的值为MaxOpenConns的`1/2`，仅供参考。

SetConnMaxLifetime

设置一个连接被使用的最长时间，即过了一段时间后会被强制回收，理论上这可以有效减少不可用连接出现的概率。当数据库方面也设置了连接的超时时间时，这个值应当不超过数据库的超时参数值。

main.go

```
package main

import (
    "github.com/gin-gonic/gin"
```

```
    "k8s-demo/config"
    "k8s-demo/controller"
    "k8s-demo/db"
    "k8s-demo/service"
)

func main() {
    //初始化k8s clientset
    service.K8s.Init()
    //初始化数据库
    db.Init()
    //初始化路由配置
    r := gin.Default()
    //初始化路由
    controller.Router.InitApiRouter(r)

    //http server启动
    r.Run(config.ListenAddr)

    //关闭db连接
    db.Close()
}
```

(2) 建立表的映射关系

model/workflow.go

```
package model

import "time"

//定义结构体，属性与mysql表字段对齐
type Workflow struct {
    //gorm:"primaryKey"用于声明主键
    ID        uint         `json:"id" gorm:"primaryKey"`
    CreatedAt *time.Time   `json:"created_at"`
    UpdatedAt *time.Time   `json:"updated_at"`
    DeletedAt *time.Time   `json:"deleted_at"`

    Name        string `json:"name"`
    Namespace   string `json:"namespace"`
    Replicas    int32  `json:"replicas"`
    Deployment  string `json:"deployment"`
    Service     string `json:"service"`
    Ingress     string `json:"ingress"`
    Type string `json:"type" gorm:"column:type"`
    //Type: clusterip nodeport ingress
}
//定义TableName方法，返回mysql表名，以此来定义mysql中的表名
func(*Workflow) TableName() string {
    return "workflow"
}
```

(3) 数据库创建表

```
CREATE TABLE `workflow` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(32) COLLATE utf8mb4_general_ci NOT NULL,
```

```
  `namespace` varchar(32) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `replicas` int DEFAULT NULL,
  `deployment` varchar(32) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `service` varchar(32) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `ingress` varchar(32) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `type` varchar(32) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `created_at` datetime DEFAULT NULL,
  `updated_at` datetime DEFAULT NULL,
  `deleted_at` datetime DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE,
  UNIQUE KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci
```

(4) 表数据列表

```
package dao

import (
    "errors"
    "github.com/wonderivan/logger"
    "k8s-demo/db"
    "k8s-demo/model"
)

var Workflow workflow

type workflow struct{}

//定义列表的返回内容，Items是workflow元素列表，Total为workflow元素数量
type WorkflowResp struct {
    Items []*model.Workflow        `json:"items"`
    Total int                `json:"total"`
}

//获取列表分页查询
func(w *workflow) GetList(name string, page, limit int) (data *WorkflowResp, err
error) {
    //定义分页数据的起始位置
    startSet := (page-1) * limit

    //定义数据库查询返回内容
    var workflowList []*model.Workflow

    //数据库查询，Limit方法用于限制条数，Offset方法设置起始位置
    tx := db.GORM.
        Where("name like ?", "%" + name + "%").
        Limit(limit).
        Offset(startSet).
        Order("id desc").
        Find(&workflowList)
    //gorm会默认把空数据也放到err中，故这里要排除空数据的情况
    if tx.Error != nil && tx.Error.Error() != "record not found" {
        logger.Error("获取Workflow列表失败, " + tx.Error.Error())
        return nil, errors.New("获取Workflow列表失败, " + tx.Error.Error())
    }

    return &WorkflowResp{
        Items: workflowList,
```

```
        Total: len(workflowList),
    }, nil
}
```

(5) 获取单条

```
//查询workflow单条数据
func(w *workflow) GetById(id int) (workflow *model.Workflow, err error) {
    workflow = &model.Workflow{}
    tx := db.GORM.Where("id = ?", id).First(&workflow)
    if tx.Error != nil && tx.Error.Error() != "record not found" {
        logger.Error("获取Workflow单条数据失败, " + tx.Error.Error())
        return nil, errors.New("获取Workflow单条数据失败, " + tx.Error.Error())
    }
    return
}
```

(6) 表数据新增

```
//新增workflow
func(w *workflow) Add(workflow *model.Workflow) (err error) {
    tx := db.GORM.Create(&workflow)
    if tx.Error != nil {
        logger.Error("添加Workflow失败, " + tx.Error.Error())
        return errors.New("添加Workflow失败, " + tx.Error.Error())
    }
    return nil
}
```

(7) 表数据删除

```
//删除workflow
//软删除 db.GORM.Delete("id = ?", id)
//软删除执行的是UPDATE语句，将deleted_at字段设置为时间即可，gorm 默认就是软删。
//实际执行语句 UPDATE `workflow` SET `deleted_at` = '2021-03-01 08:32:11' WHERE `id` IN
('1'
//硬删除 db.GORM.Unscoped().Delete("id = ?", id)) 直接从表中删除这条数据
//实际执行语句 DELETE FROM `workflow` WHERE `id` IN ('1');
func(w *workflow) DelById(id int) (err error) {
    tx := db.GORM.Where("id = ?", id).Delete(&model.Workflow{})
    if tx.Error != nil {
        logger.Error("删除Workflow失败, " + tx.Error.Error())
        return errors.New("删除Workflow失败, " + tx.Error.Error())
    }
    return nil
}
```

6.3 Workflow

service/workflow.go

(1) 列表

```
//获取列表分页查询
func(w *workflow) GetList(name string, page, limit int) (data *dao.WorkflowResp, err
error) {
    data, err = dao.Workflow.GetList(name, page, limit)
    if err != nil {
        return nil, err
    }
    return data, nil
}
```

(2) 获取Workflow详情

```
//查询workflow单条数据
func(w *workflow) GetById(id int) (data *model.Workflow, err error) {
    data, err = dao.Workflow.GetById(id)
    if err != nil {
        return nil, err
    }
    return data, nil
}
```

(3) 新增Workflow

```
//定义WorkflowCreate结构体，用于创建workflow需要的参数属性的定义
type WorkflowCreate struct {
    Name           string  `json:"name"`
    Namespace      string  `json:"namespace"`
    Replicas       int32   `json:"replicas"`
    Image          string  `json:"image"`
    Label          map[string]string  `json:"label"`
    Cpu            string  `json:"cpu"`
    Memory         string  `json:"memory"`
    ContainerPort  int32   `json:"container_port"`
    HealthCheck    bool    `json:"health_check"`
    HealthPath     string  `json:"health_path"`
    Type           string  `json:"type"`
    Port           int32   `json:"port"`
    NodePort       int32   `json:"node_port"`
    Hosts          map[string][]*HttpPath `json:"hosts"`
}

//创建workflow
func(w *workflow) CreateWorkflow(data *WorkflowCreate) (err error) {
    //若workflow不是ingress类型，传入空字符串即可
    var ingressName string
    if data.Type == "Ingress" {
        ingressName = getIngressName(data.Name)
    } else {
        ingressName = ""
    }
    //组装mysql中workflow的单条数据
    workflow := &model.Workflow{
        Name:        data.Name,
        Namespace:   data.Namespace,
        Replicas:    data.Replicas,
        Deployment:  data.Name,
        Service:     getServiceName(data.Name),
        Ingress:     ingressName,
```

```
        Type:        data.Type,
    }
    //调用dao层执行数据库的添加操作
    err = dao.Workflow.Add(workflow)
    if err != nil {
        return err
    }

    //创建k8s资源
    err = createWorkflowRes(data)
    if err != nil {
        return err
    }

    return nil
}

//封装创建workflow对应的k8s资源
//小写开头的函数，作用域只在当前包中，不支持跨包调用
func createWorkflowRes(data *WorkflowCreate) (err error) {
    //声明service类型
    var serviceType string
    //组装DeployCreate类型的数据
    dc := &DeployCreate{
        Name:          data.Name,
        Namespace:     data.Namespace,
        Replicas:      data.Replicas,
        Image:         data.Image,
        Label:         data.Label,
        Cpu:           data.Cpu,
        Memory:        data.Memory,
        ContainerPort: data.ContainerPort,
        HealthCheck:   data.HealthCheck,
        HealthPath:    data.HealthPath,
    }
    //创建deployment
    err = Deployment.CreateDeployment(dc)
    if err != nil {
        return err
    }
    //判断service类型
    if data.Type != "Ingress" {
        serviceType = data.Type
    } else {
        serviceType = "ClusterIP"
    }

    //组装ServiceCreate类型的数据
    sc := &ServiceCreate{
        Name:          getServiceName(data.Name),
        Namespace:     data.Namespace,
        Type:          serviceType,
        ContainerPort: data.ContainerPort,
        Port:          data.Port,
        NodePort:      data.NodePort,
        Label:         data.Label,
    }
    err = Servicev1.CreateService(sc)
```

```
    if err != nil {
        return err
    }
    //组装IngressCreate类型的数据，创建ingress，只有ingress类型的workflow才有ingress资源，所以这里做了一层判断
    if data.Type == "Ingress" {
        ic := &IngressCreate{
            Name:      getIngressName(data.Name),
            Namespace: data.Namespace,
            Label:     data.Label,
            Hosts:     data.Hosts,
        }
        err = Ingress.CreateIngress(ic)
        if err != nil {
            return err
        }
    }
    return nil
}

//workflow名字转换成service名字，添加-svc后缀
func getServiceName(workflowName string) (serviceName string) {
    return workflowName + "-svc"
}
//workflow名字转换成ingress名字，添加-ing后缀
func getIngressName(workflowName string) (ingressName string) {
    return workflowName + "-ing"
}
```

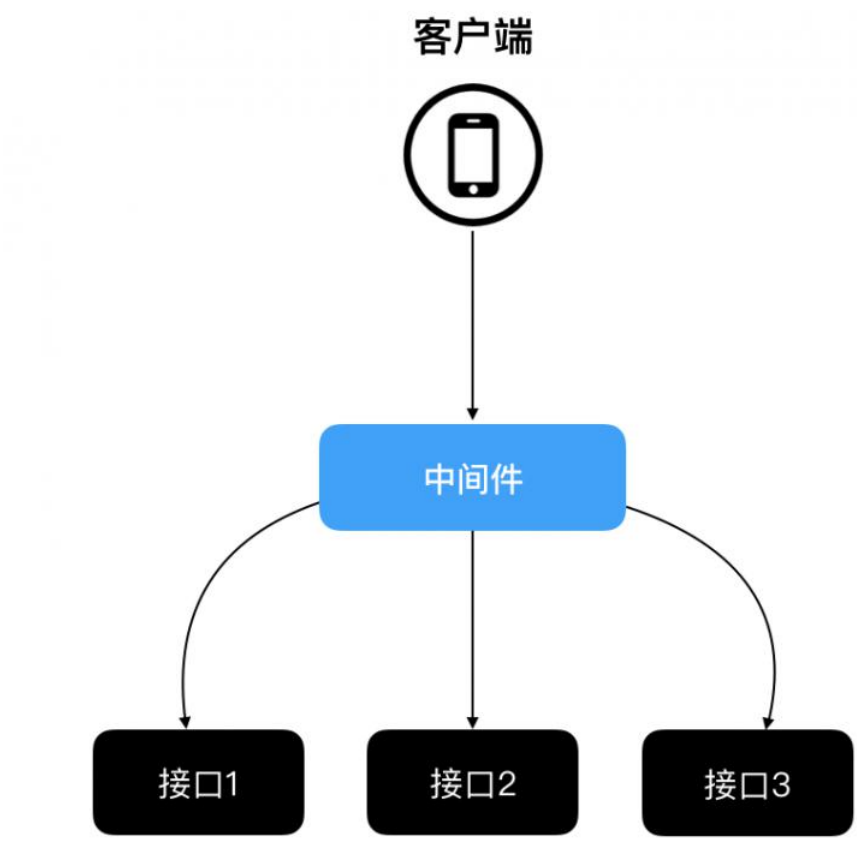
(4) 删除Workflow

```
//删除workflow
func(w *workflow) DelById(id int) (err error) {
    //获取workflow数据
    workflow, err := dao.Workflow.GetById(id)
    if err != nil {
        return err
    }
    //删除k8s资源
    err = delWorkflowRes(workflow)
    if err != nil {
        return err
    }
    //删除数据库数据
    err = dao.Workflow.DelById(id)
    if err != nil {
        return err
    }

    return nil
}

//封装删除workflow对应的k8s资源
func delWorkflowRes(workflow *model.Workflow) (err error) {
    //删除deployment
    err = Deployment.DeleteDeployment(workflow.Name, workflow.Namespace)
    if err != nil {
        return err
```

```
    }
    //删除service
    err = Servicev1.DeleteService(getServiceName(workflow.Name), workflow.Namespace)
    if err != nil {
        return err
    }
    //删除ingress，这里多了一层判断，因为只有type为ingress的workflow才有ingress资源
    if workflow.Type == "Ingress" {
        err = Ingress.DeleteIngress(getIngressName(workflow.Name), workflow.Namespace)
        if err != nil {
            return err
        }
    }

    return nil
}
```

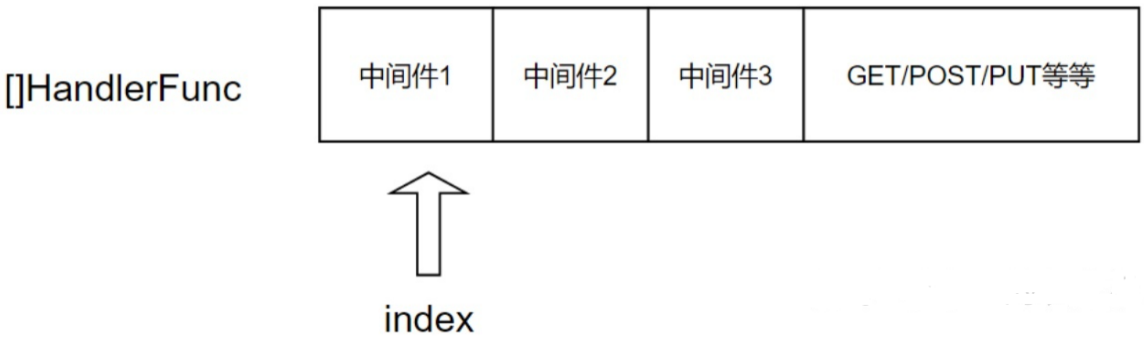
7、中间件

7.1 什么是中间件



中间件，英译middleware，顾名思义，放在中间的物件，那么放在谁中间呢？本来，客户端可以直接请求到服务端接口。现在，中间件横插一脚，它能在请求到达接口之前拦截请求，做一些特殊处理，比如日志记录，故障处理等。

7.2 gin中间件用法



因为gin的中间件函数与业务逻辑处理函数是放到gin的队列中的，所以当一个中间件函数执行return语句时只代表当前中间件函数执行完了，gin框架会驱动index++,然后执行队列中后续的中间件函数或逻辑处理函数，当在中间件函数中执行context.Next()时，gin框架也会驱动index++，执行下一个函数。当执行context.Abort()时，会修改c.index = 63.5,由于该索引不存在，所以队列中后面的的中间件函数和逻辑处理函数就不会执行了。

（1）定义一个返回值是gin.HandlerFunc的方法

（2）在方法中根据context上下文添加中间件逻辑

（3）中间件逻辑未通过，使用context.Abort()和return停止下个函数的执行

（4）中间件逻辑通过时，使用context.Next()继续执行下个函数

（5）定义好中间件函数后，在main中使用use()将其加入到队列中，注意use一定要在初始化路由的前面，否则不会生效

7.2 Cors跨域

代码层直接处理跨域请求，不需要前面再加一层nginx处理，解决前后端域名不同、IP不同甚至端口不同导致的跨域报错。

middle/cors.go

```
package middle

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

//处理跨域请求,支持options访问
func Cors() gin.HandlerFunc {
    return func(c *gin.Context) {
        //获取请求方法
        method := c.Request.Method

        //添加跨域响应头
        c.Header("Content-Type", "application/json")
        c.Header("Access-Control-Allow-Origin", "*")
        c.Header("Access-Control-Max-Age", "86400")
        c.Header("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE,
UPDATE")
        c.Header("Access-Control-Allow-Headers", "X-Token, Content-Type, Content-
Length, Accept-Encoding, X-CSRF-Token, Authorization, X-Max")
        c.Header("Access-Control-Allow-Credentials", "false")
```

```
        //放行所有OPTIONS方法
        if method == "OPTIONS" {
            c.AbortWithStatus(http.StatusNoContent)
        }
        //处理请求
        c.Next()
    }
}
```

main.go

```
package main

import (
    "github.com/gin-gonic/gin"
    "k8s-demo/config"
    "k8s-demo/controller"
    "k8s-demo/db"
    "k8s-demo/middle"
    "k8s-demo/service"
)

func main() {
    //初始化k8s clientset
    service.K8s.Init()
    //初始化数据库
    db.Init()
    //初始化路由配置
    r := gin.Default()
    //跨域配置
    r.Use(middle.Cors())
    //初始化路由
    controller.Router.InitApiRouter(r)

    //http server启动
    r.Run(config.ListenAddr)
}
```

7.3 JWT token验证

验证请求的合法性，前端只有在登录状态下才会生成token，请求时将token放入Header中，后端接收的请求时，先由该中间件验证token是否合法，合法时才放行，继续执行业务函数的逻辑处理。

utils/jwt.go

```
package utils

import (
    "errors"
    "github.com/dgrijalva/jwt-go"
    "github.com/wonderivan/logger"
)

var JWTToken jwtToken

type jwtToken struct{}
```

```
//token中包含的自定义信息以及jwt签名信息
type CustomClaims struct {
    UserName  string `json:"username"`
    Password  string `json:"password"`
    jwt.StandardClaims
}

//加解密因子
const (
    SECRET = "adoodevops"
)

//解析token
func (*jwtToken) ParseToken(tokenString string) (claims *CustomClaims, err error) {
    //使用jwt.ParseWithClaims方法解析token，这个token是前端传给我们的,获得一个*Token类型的对象
    token, err := jwt.ParseWithClaims(tokenString, &CustomClaims{}, func(token
*jwt.Token) (interface{}, error) {
        return []byte(SECRET), nil
    })
    if err != nil {
        logger.Error("parse token failed ", err)
        //处理token解析后的各种错误
        if ve, ok := err.(*jwt.ValidationError); ok {
            if ve.Errors&jwt.ValidationErrorMalformed != 0 {
                return nil, errors.New("TokenMalformed")
            } else if ve.Errors&jwt.ValidationErrorExpired != 0 {
                return nil, errors.New("TokenExpired")
            } else if ve.Errors&jwt.ValidationErrorNotValidYet != 0 {
                return nil, errors.New("TokenNotValidYet")
            } else {
                return nil, errors.New("TokenInvalid")
            }
        }
    }
    //转换成*CustomClaims类型并返回
    if claims, ok := token.Claims.(*CustomClaims); ok && token.Valid {
        return claims, nil
    }
    return nil, errors.New("解析Token失败")
}
```

middle/jwt.go

```
package middle

import (
    "github.com/gin-gonic/gin"
    "k8s-demo/utils"
    "net/http"
)

// JWTAuth 中间件，检查token
func JWTAuth() gin.HandlerFunc {
    return func(c *gin.Context) {
        //对登录接口放行
        if (len(c.Request.URL.String()) >= 10 && c.Request.URL.String()[0:10] ==
"/api/login") {
```

```
            c.Next()
        } else {
            //获取Header中的Authorization
            token := c.Request.Header.Get("Authorization")
            if token == "" {
                c.JSON(http.StatusBadRequest, gin.H{
                    "msg": "请求未携带token，无权限访问",
                    "data": nil,
                })
                c.Abort()
                return
            }

            // parseToken 解析token包含的信息
            claims, err := utils.JWTToken.ParseToken(token)
            if err != nil {
                //token延期错误
                if err.Error() == "TokenExpired" {
                    c.JSON(http.StatusBadRequest, gin.H{
                        "msg": "授权已过期",
                        "data": nil,
                    })
                    c.Abort()
                    return
                }
                //其他解析错误
                c.JSON(http.StatusBadRequest, gin.H{
                    "msg": err.Error(),
                    "data": nil,
                })
                c.Abort()
                return
            }
            // 继续交由下一个路由处理,并将解析出的信息传递下去
            c.Set("claims", claims)

            c.Next()
        }
    }
}
```

main.go

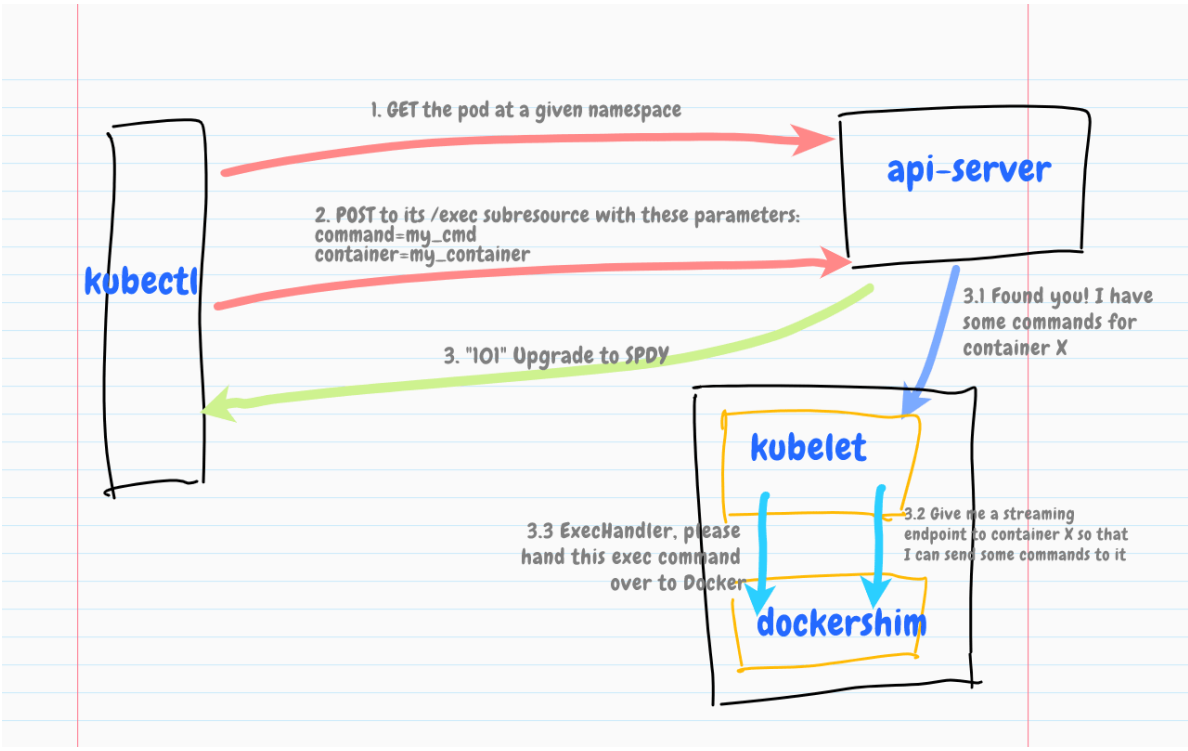
```
package main

import (
    "github.com/gin-gonic/gin"
    "k8s-demo/config"
    "k8s-demo/controller"
    "k8s-demo/db"
    "k8s-demo/middle"
    "k8s-demo/service"
)

func main() {
    //初始化k8s clientset
    service.K8s.Init()
```

```
    //初始化数据库
    db.Init()
    //初始化路由配置
    r := gin.Default()
    //跨域配置
    r.Use(middle.Cors())
    //jwt token验证
    r.Use(middle.JWTAuth())
    //初始化路由
    controller.Router.InitApiRouter(r)

    //http server启动
    r.Run(config.ListenAddr)
}
```

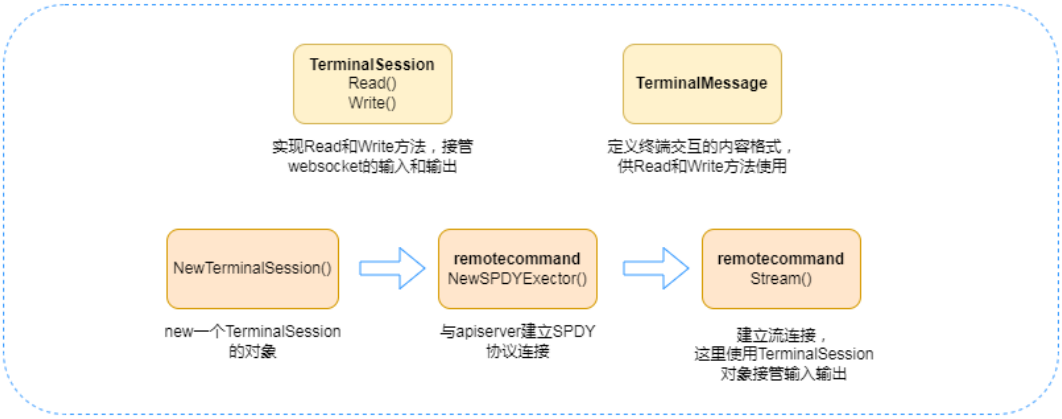
8、WebShell终端

> 阿良教育： www.aliangedu.cn

8.1 kubectl exec 原理



8.2 实现思路



通过 client-go 提供的方法，实现通过网页进入 kubernetes pod 的终端操作。

- client-go remotecommand
- websocket
- xterm.js

remotecommand

`k8s.io/client-go/tools/remotecommand` kubernetes client-go 提供的 remotecommand 包，提供了方法与集群中的容器建立长连接，并设置容器的 stdin，stdout 等。
remotecommand 包提供基于 `SPDY` 协议的 Executor interface，进行和 pod 终端的流的传输。初始化一个 Executor 很简单，只需要调用 remotecommand 的 NewSPDYExecutor 并传入对应参数。
Executor 的 Stream 方法，会建立一个流传输的连接，直到服务端和调用端一端关闭连接，才会停止传输。常用的做法是定义一个如下 `PtyHandler` 的 interface，然后使用你想用的客户端实现该 interface 对应的 `Read(p []byte) (int, error)` 和 `Write(p []byte) (int, error)` 方法即可，调用 Stream 方法时，只要将 StreamOptions 的 Stdin Stdout 都设置为 ptyHandler，Executor 就会通过你定义的 write 和 read 方法来传输数据。

websocket

`github.com/gorilla/websocket` 是 go 的一个 websocket 实现，提供了全面的 websocket 相关的方法，这里使用它来实现上面所说的 `PtyHandler` 接口。
首先定义一个 TerminalSession 类，该类包含一个 `*websocket.Conn`，通过 websocket 连接实现 `PtyHandler` 接口的读写方法，Next 方法在 remotecommand 执行过程中会被调用。

xterm.js

前端页面使用 `xterm.js` 进行模拟terminal展示，只要 javascript 监听 Terminal 对象的对应事件及 websocket 连接的事件，进行对应的页面展示和消息推送就可以了。

8.3 代码实现

（1）处理终端交互

service/terminal.go

```
package service

import (
    "encoding/json"
    "fmt"
    "github.com/gorilla/websocket"
    "github.com/wonderivan/logger"
```

```
    "k8s-demo/config"
    v1 "k8s.io/api/core/v1"
    "k8s.io/client-go/kubernetes/scheme"
    "k8s.io/client-go/tools/clientcmd"
    "k8s.io/client-go/tools/remotecommand"
    "log"
    "net/http"
    "time"
)

var Terminal terminal

type terminal struct {}


//定义websocket的handler方法
func(t *terminal) WsHandler(w http.ResponseWriter, r *http.Request) {
    //加载k8s配置
    conf, err := clientcmd.BuildConfigFromFlags("", config.Kubeconfig)
    if err != nil {
        logger.Error("创建k8s配置失败, " + err.Error())
    }
    //解析form入参，获取namespace、podName、containerName参数
    if err := r.ParseForm(); err != nil {
        return
    }
    namespace := r.Form.Get("namespace")
    podName := r.Form.Get("pod_name")
    containerName := r.Form.Get("container_name")
    logger.Info("exec pod: %s, container: %s, namespace: %s\n", podName,
containerName, namespace)
    //new一个TerminalSession类型的pty实例
    pty, err := NewTerminalSession(w, r, nil)
    if err != nil {
        logger.Error("get pty failed: %v\n", err)
        return
    }
    //处理关闭
    defer func() {
        logger.Info("close session.")
        pty.Close()
    }()
    // 初始化pod所在的corev1资源组
    // PodExecOptions struct 包括Container stdout stdout  Command 等结构
    // scheme.ParameterCodec 应该是pod 的GVK （GroupVersion & Kind）之类的
    // URL长相:
    // https://192.168.1.11:6443/api/v1/namespaces/default/pods/nginx-wf2-778d88d7c-
7rmsk/exec?command=%2Fbin%2Fbash&container=nginx-
wf2&stderr=true&stdin=true&stdout=true&tty=true
    req := K8s.ClientSet.CoreV1().RESTClient().Post().
        Resource("pods").
        Name(podName).
        Namespace(namespace).
        SubResource("exec").
        VersionedParams(&v1.PodExecOptions{
            Container: containerName,
            Command:   []string{"/bin/bash"},
            Stdin:     true,
```

```
            Stdout:    true,
            Stderr:    true,
            TTY:       true,
        }, scheme.ParameterCodec)
    fmt.Println(req.URL())

    //remotecommand 主要实现了http 转 SPDY 添加X-Stream-Protocol-Version相关header 并发送请
求
    executor, err := remotecommand.NewSPDYExecutor(conf,"POST", req.URL())
    if err != nil {
        return
    }
    // 建立链接之后从请求的sream中发送、读取数据
    err = executor.Stream(remotecommand.StreamOptions{
        Stdin:             pty,
        Stdout:            pty,
        Stderr:            pty,
        TerminalSizeQueue: pty,
        Tty:               true,
    })

    if err != nil {
        msg := fmt.Sprintf("Exec to pod error! err: %v", err)
        logger.Info(msg)
        //将报错返回出去
        pty.Write([]byte(msg))
        //标记退出stream流
        pty.Done()
    }
}

const END_OF_TRANSMISSION = "\u0004"

//TerminalMessage定义了终端和容器shell交互内容的格式
//Operation是操作类型
//Data是具体数据内容
//Rows和Cols可以理解为终端的行数和列数，也就是宽、高
type TerminalMessage struct {
    Operation string `json:"operation"`
    Data      string `json:"data"`
    Rows      uint16 `json:"rows"`
    Cols      uint16 `json:"cols"`
}

//初始化一个websocket.Upgrader类型的对象，用于http协议升级为websocket协议
var upgrader = func() websocket.Upgrader {
    upgrader := websocket.Upgrader{}
    upgrader.HandshakeTimeout = time.Second * 2
    upgrader.CheckOrigin = func(r *http.Request) bool {
        return true
    }
    return upgrader
}()

//定义TerminalSession结构体，实现PtyHandler接口
//wsConn是websocket连接
//sizeChan用来定义终端输入和输出的宽和高
//doneChan用于标记退出终端
```

```go
type TerminalSession struct {
    wsConn   *websocket.Conn
    sizeChan chan remotecommand.TerminalSize
    doneChan chan struct{}
}

//该方法用于升级http协议至websocket，并new一个TerminalSession类型的对象返回
func NewTerminalSession(w http.ResponseWriter, r *http.Request, responseHeader
http.Header) (*TerminalSession, error) {
    conn, err := upgrader.Upgrade(w, r, responseHeader)
    if err != nil {
        return nil, err
    }
    session := &TerminalSession{
        wsConn:   conn,
        sizeChan: make(chan remotecommand.TerminalSize),
        doneChan: make(chan struct{}),
    }
    return session, nil
}

// 关闭doneChan，关闭后触发退出终端
func (t *TerminalSession) Done() {
    close(t.doneChan)
}

//获取web端是否resize，以及是否退出终端
func (t *TerminalSession) Next() *remotecommand.TerminalSize {
    select {
    case size := <-t.sizeChan:
        return &size
    case <-t.doneChan:
        return nil
    }
}

//用于读取web端的输入，接收web端输入的指令内容
func (t *TerminalSession) Read(p []byte) (int, error) {
    _, message, err := t.wsConn.ReadMessage()

    if err != nil {
        log.Printf("read message err: %v", err)
        return copy(p, END_OF_TRANSMISSION), err
    }
    var msg TerminalMessage
    if err := json.Unmarshal([]byte(message), &msg); err != nil {
        log.Printf("read parse message err: %v", err)
        // return 0, nil
        return copy(p, END_OF_TRANSMISSION), err
    }

    switch msg.Operation {
    case "stdin":
        return copy(p, msg.Data), nil
    case "resize":
        t.sizeChan <- remotecommand.TerminalSize{Width: msg.Cols, Height: msg.Rows}
        return 0, nil
    case "ping":
```

```
            return 0, nil
        default:
            log.Printf("unknown message type '%s'", msg.Operation)
            // return 0, nil
            return copy(p, END_OF_TRANSMISSION), fmt.Errorf("unknown message type '%s'",
msg.Operation)
        }
}

//用于向web端输出，接收web端的指令后，将结果返回出去
func (t *TerminalSession) Write(p []byte) (int, error) {
    msg, err := json.Marshal(TerminalMessage{
        Operation: "stdout",
        Data:      string(p),
    })
    if err != nil {
        log.Printf("write parse message err: %v", err)
        return 0, err
    }
    if err := t.wsConn.WriteMessage(websocket.TextMessage, msg); err != nil {
        log.Printf("write message err: %v", err)
        return 0, err
    }
    return len(p), nil
}

// 用于关闭websocket连接
func (t *TerminalSession) Close() error {
    return t.wsConn.Close()
}
```

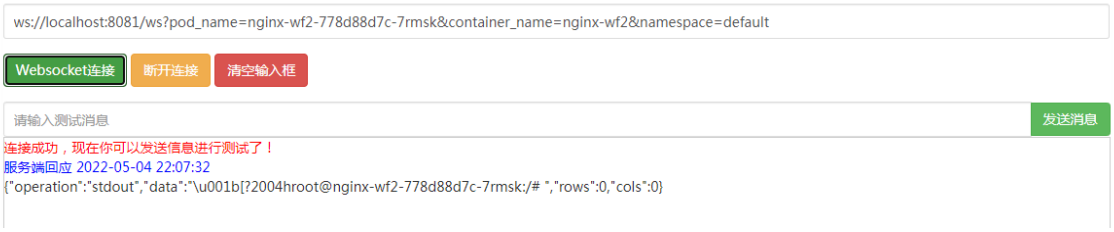
（2）由于会将http升级为websocket协议，故需要重新监听个端口

main.go

```
package main

import (
    "github.com/gin-gonic/gin"
    "k8s-demo/config"
    "k8s-demo/controller"
    "k8s-demo/db"
    "k8s-demo/middle"
    "k8s-demo/service"
    "net/http"
)

func main() {
    //初始化k8s clientset
    service.K8s.Init()
    //初始化数据库
    db.Init()
    //初始化路由配置
    r := gin.Default()
    //跨域配置
    r.Use(middle.Cors())
    //jwt token验证
    //r.Use(middle.JWTAuth())
```

```
    //初始化路由
    controller.Router.InitApiRouter(r)

    //终端websocket
    go func() {
        http.HandleFunc("/ws", service.Terminal.WsHandler)
        http.ListenAndServe(":8081", nil)
    }()

    //http server启动
    r.Run(config.ListenAddr)

    //关闭db连接
    db.Close()
}
```

（3）websocket测试



9、总结

至此，K8s管理系统后端代码开发完毕。基本上开发的内容都是k8s中的原生功能，没有较为复杂的代码逻辑。本节课程旨在借助K8s项目，让大家逐渐掌握开发思路与技巧，做一个go+gin项目开发的实战入门。希望大家在完成本节课程的学习后，能够独立完成脚本/接口的开发，以及基于此项目开发更多的新功能。在掌握此项技能后，你回发现以前不能做的需求，现在能做了，以前看不懂的内容，现在看懂了，能力更强了，升职加薪，走上人生巅峰！