# 源码导出文档

package middleware  
  
import (  
 "fmt"  
 "strings"  
 "time"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
)  
  
// 操作日志channel  
var OperationLogChan = make(chan \*model.OperationLog, 30)  
  
func OperationLogMiddleware() gin.HandlerFunc {  
 return func(c \*gin.Context) {  
 // 开始时间  
 startTime := time.Now()  
  
 // 处理请求  
 c.Next()  
  
 // 结束时间  
 endTime := time.Now()  
  
 // 执行耗时  
 timeCost := endTime.Sub(startTime).Milliseconds()  
  
 // 获取当前登录用户  
 var username string  
 ctxUser, \_ := c.Get("user")  
 user, ok := ctxUser.(model.User)  
 if !ok {  
 username = "未登录"  
 } else {  
 username = user.Username  
 }  
  
 // 获取访问路径  
 path := strings.TrimPrefix(c.FullPath(), "/"+config.Conf.System.UrlPathPrefix)  
 // 请求方式  
 method := c.Request.Method  
  
 // 获取接口描述  
 api := new(model.Api)  
 \_ = isql.Api.Find(tools.H{"path": path, "method": method}, api)  
  
 operationLog := model.OperationLog{  
 Username: username,  
 Ip: c.ClientIP(),  
 IpLocation: "",  
 Method: method,  
 Path: path,  
 Remark: api.Remark,  
 Status: c.Writer.Status(),  
 StartTime: fmt.Sprintf("%v", startTime),  
 TimeCost: timeCost,  
 UserAgent: c.Request.UserAgent(),  
 }  
  
 // 最好是将日志发送到rabbitmq或者kafka中  
 // 这里是发送到channel中，开启3个goroutine处理  
 OperationLogChan <- &operationLog  
 }  
}

package middleware  
  
import (  
 "fmt"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/common"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "time"  
  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/model/response"  
  
 jwt "github.com/appleboy/gin-jwt/v2"  
 "github.com/gin-gonic/gin"  
)  
  
// 初始化jwt中间件  
func InitAuth() (\*jwt.GinJWTMiddleware, error) {  
 authMiddleware, err := jwt.New(&jwt.GinJWTMiddleware{  
 Realm: config.Conf.Jwt.Realm, // jwt标识  
 Key: []byte(config.Conf.Jwt.Key), // 服务端密钥  
 Timeout: time.Hour \* time.Duration(config.Conf.Jwt.Timeout), // token过期时间  
 MaxRefresh: time.Hour \* time.Duration(config.Conf.Jwt.MaxRefresh), // token最大刷新时间(RefreshToken过期时间=Timeout+MaxRefresh)  
 PayloadFunc: payloadFunc, // 有效载荷处理  
 IdentityHandler: identityHandler, // 解析Claims  
 Authenticator: login, // 校验token的正确性, 处理登录逻辑  
 Authorizator: authorizator, // 用户登录校验成功处理  
 Unauthorized: unauthorized, // 用户登录校验失败处理  
 LoginResponse: loginResponse, // 登录成功后的响应  
 LogoutResponse: logoutResponse, // 登出后的响应  
 RefreshResponse: refreshResponse, // 刷新token后的响应  
 TokenLookup: "header: Authorization, query: token, cookie: jwt", // 自动在这几个地方寻找请求中的token  
 TokenHeadName: "Bearer", // header名称  
 TimeFunc: time.Now,  
 })  
 return authMiddleware, err  
}  
  
// 有效载荷处理  
func payloadFunc(data interface{}) jwt.MapClaims {  
 if v, ok := data.(tools.H); ok {  
 var user model.User  
 // 将用户json转为结构体  
 tools.JsonI2Struct(v["user"], &user)  
 return jwt.MapClaims{  
 jwt.IdentityKey: user.ID,  
 "user": v["user"],  
 }  
 }  
 return jwt.MapClaims{}  
}  
  
// 解析Claims  
func identityHandler(c \*gin.Context) interface{} {  
 claims := jwt.ExtractClaims(c)  
 // 此处返回值类型map[string]interface{}与payloadFunc和authorizator的data类型必须一致, 否则会导致授权失败还不容易找到原因  
 return tools.H{  
 "IdentityKey": claims[jwt.IdentityKey],  
 "user": claims["user"],  
 }  
}  
  
// 校验token的正确性, 处理登录逻辑  
func login(c \*gin.Context) (interface{}, error) {  
 var req request.RegisterAndLoginReq  
 // 请求json绑定  
 if err := c.ShouldBind(&req); err != nil {  
 return "", err  
 }  
  
 // 密码通过RSA解密  
 decodeData, err := tools.RSADecrypt([]byte(req.Password), config.Conf.System.RSAPrivateBytes)  
 if err != nil {  
 return nil, err  
 }  
  
 u := &model.User{  
 Username: req.Username,  
 Password: string(decodeData),  
 }  
  
 // 密码校验  
 user, err := isql.User.Login(u)  
 if err != nil {  
 return nil, err  
 }  
 // 将用户以json格式写入, payloadFunc/authorizator会使用到  
 return tools.H{  
 "user": tools.Struct2Json(user),  
 }, nil  
}  
  
// 用户登录校验成功处理  
func authorizator(data interface{}, c \*gin.Context) bool {  
 if v, ok := data.(tools.H); ok {  
 userStr := v["user"].(string)  
 var user model.User  
 // 将用户json转为结构体  
 tools.Json2Struct(userStr, &user)  
 // 将用户保存到context, api调用时取数据方便  
 c.Set("user", user)  
 return true  
 }  
 return false  
}  
  
// 用户登录校验失败处理  
func unauthorized(c \*gin.Context, code int, message string) {  
 common.Log.Debugf("JWT认证失败, 错误码: %d, 错误信息: %s", code, message)  
 response.Response(c, code, code, nil, fmt.Sprintf("JWT认证失败, 错误码: %d, 错误信息: %s", code, message))  
}  
  
// 登录成功后的响应  
func loginResponse(c \*gin.Context, code int, token string, expires time.Time) {  
 response.Response(c, code, code,  
 gin.H{  
 "token": token,  
 "expires": expires.Format("2006-01-02 15:04:05"),  
 },  
 "登录成功")  
}  
  
// 登出后的响应  
func logoutResponse(c \*gin.Context, code int) {  
 response.Success(c, nil, "退出成功")  
}  
  
// 刷新token后的响应  
func refreshResponse(c \*gin.Context, code int, token string, expires time.Time) {  
 response.Response(c, code, code,  
 gin.H{  
 "token": token,  
 "expires": expires,  
 },  
 "刷新token成功")  
}

package middleware  
  
import (  
 "strings"  
 "sync"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/public/common"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
)  
  
var checkLock sync.Mutex  
  
// Casbin中间件, 基于RBAC的权限访问控制模型  
func CasbinMiddleware() gin.HandlerFunc {  
 return func(c \*gin.Context) {  
 user, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 tools.Response(c, 401, 401, nil, "用户未登录")  
 c.Abort()  
 return  
 }  
 if user.Status != 1 {  
 tools.Response(c, 401, 401, nil, "当前用户已被禁用")  
 c.Abort()  
 return  
 }  
 // 获得用户的全部角色  
 roles := user.Roles  
 // 获得用户全部未被禁用的角色的Keyword  
 var subs []string  
 for \_, role := range roles {  
 if role.Status == 1 {  
 subs = append(subs, role.Keyword)  
 }  
 }  
 // 获得请求路径URL  
 obj := strings.TrimPrefix(c.FullPath(), "/"+config.Conf.System.UrlPathPrefix)  
 // 获取请求方式  
 act := c.Request.Method  
 isPass := check(subs, obj, act)  
 if !isPass {  
 tools.Response(c, 401, 401, nil, "没有权限")  
 c.Abort()  
 return  
 }  
  
 c.Next()  
 }  
}  
  
func check(subs []string, obj string, act string) bool {  
 // 同一时间只允许一个请求执行校验, 否则可能会校验失败  
 checkLock.Lock()  
 defer checkLock.Unlock()  
 isPass := false  
 for \_, sub := range subs {  
 pass, \_ := common.CasbinEnforcer.Enforce(sub, obj, act)  
 if pass {  
 isPass = true  
 break  
 }  
 }  
 return isPass  
}

package middleware  
  
import (  
 "fmt"  
 "net/http"  
 "strings"  
  
 "github.com/gin-gonic/gin"  
)  
  
// CORS跨域中间件  
func CORSMiddleware() gin.HandlerFunc {  
 return func(c \*gin.Context) {  
 method := c.Request.Method //请求方法  
 origin := c.Request.Header.Get("Origin") //请求头部  
 var headerKeys []string // 声明请求头keys  
 for k := range c.Request.Header {  
 headerKeys = append(headerKeys, k)  
 }  
 headerStr := strings.Join(headerKeys, ", ")  
 if headerStr != "" {  
 headerStr = fmt.Sprintf("access-control-allow-origin, access-control-allow-headers, %s", headerStr)  
 } else {  
 headerStr = "access-control-allow-origin, access-control-allow-headers"  
 }  
 if origin != "" {  
 c.Writer.Header().Set("Access-Control-Allow-Origin", "\*")  
 c.Header("Access-Control-Allow-Origin", "\*") // 这是允许访问所有域  
 c.Header("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE,UPDATE") //服务器支持的所有跨域请求的方法,为了避免浏览次请求的多次'预检'请求  
 // header的类型  
 c.Header("Access-Control-Allow-Headers", "Authorization, Content-Length, X-CSRF-Token, Token,session,X\_Requested\_With,Accept, Origin, Host, Connection, Accept-Encoding, Accept-Language,DNT, X-CustomHeader, Keep-Alive, User-Agent, X-Requested-With, If-Modified-Since, Cache-Control, Content-Type, Pragma")  
 // 允许跨域设置 可以返回其他子段  
 c.Header("Access-Control-Expose-Headers", "Content-Length, Access-Control-Allow-Origin, Access-Control-Allow-Headers,Cache-Control,Content-Language,Content-Type,Expires,Last-Modified,Pragma,FooBar") // 跨域关键设置 让浏览器可以解析  
 c.Header("Access-Control-Max-Age", "172800") // 缓存请求信息 单位为秒  
 c.Header("Access-Control-Allow-Credentials", "false") // 跨域请求是否需要带cookie信息 默认设置为true  
 c.Set("content-type", "application/json") // 设置返回格式是json  
 }  
 //放行所有OPTIONS方法  
 if method == "OPTIONS" {  
 c.JSON(http.StatusOK, "Options Request!")  
 }  
 // 处理请求  
 c.Next() // 处理请求  
 \_ = headerStr  
 }  
}

package middleware  
  
import (  
 "time"  
  
 "github.com/eryajf/go-ldap-admin/model/response"  
  
 "github.com/gin-gonic/gin"  
 "github.com/juju/ratelimit"  
)  
  
func RateLimitMiddleware(fillInterval time.Duration, capacity int64) gin.HandlerFunc {  
 bucket := ratelimit.NewBucket(fillInterval, capacity)  
 return func(c \*gin.Context) {  
 if bucket.TakeAvailable(1) < 1 {  
 response.Fail(c, nil, "访问限流")  
 c.Abort()  
 return  
 }  
 c.Next()  
 }  
}

package middleware  
  
import (  
 "embed"  
 "io/fs"  
 "net/http"  
 "os"  
 "path"  
 "strings"  
  
 "github.com/gin-gonic/gin"  
)  
  
const INDEX = "index.html"  
  
type ServeFileSystem interface {  
 http.FileSystem  
 Exists(prefix string, path string) bool  
}  
  
type localFileSystem struct {  
 http.FileSystem  
 root string  
 indexes bool  
}  
  
func LocalFile(root string, indexes bool) \*localFileSystem {  
 return &localFileSystem{  
 FileSystem: gin.Dir(root, indexes),  
 root: root,  
 indexes: indexes,  
 }  
}  
  
func (l \*localFileSystem) Exists(prefix string, filepath string) bool {  
 if p := strings.TrimPrefix(filepath, prefix); len(p) < len(filepath) {  
 name := path.Join(l.root, p)  
 stats, err := os.Stat(name)  
 if err != nil {  
 return false  
 }  
 if stats.IsDir() {  
 if !l.indexes {  
 index := path.Join(name, INDEX)  
 \_, err := os.Stat(index)  
 if err != nil {  
 return false  
 }  
 }  
 }  
 return true  
 }  
 return false  
}  
  
func ServeRoot(urlPrefix, root string) gin.HandlerFunc {  
 return Serve(urlPrefix, LocalFile(root, false))  
}  
  
// Static returns a middleware handler that serves static files in the given directory.  
func Serve(urlPrefix string, fs ServeFileSystem) gin.HandlerFunc {  
 fileserver := http.FileServer(fs)  
 if urlPrefix != "" {  
 fileserver = http.StripPrefix(urlPrefix, fileserver)  
 }  
 return func(c \*gin.Context) {  
 if fs.Exists(urlPrefix, c.Request.URL.Path) {  
 fileserver.ServeHTTP(c.Writer, c.Request)  
 c.Abort()  
 }  
 }  
}  
  
type embedFileSystem struct {  
 http.FileSystem  
}  
  
func (e embedFileSystem) Exists(prefix string, path string) bool {  
 \_, err := e.Open(path)  
 return err == nil  
}  
  
func EmbedFolder(fsEmbed embed.FS, targetPath string) ServeFileSystem {  
 fsys, err := fs.Sub(fsEmbed, targetPath)  
 if err != nil {  
 panic(err)  
 }  
 return embedFileSystem{  
 FileSystem: http.FS(fsys),  
 }  
}

package logic  
  
import (  
 "fmt"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/model/response"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
)  
  
type OperationLogLogic struct{}  
  
// List 数据列表  
func (l OperationLogLogic) List(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.OperationLogListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 fmt.Println(r)  
 // 获取数据列表  
 logs, err := isql.OperationLog.List(r)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取接口列表失败: %s", err.Error()))  
 }  
  
 rets := make([]model.OperationLog, 0)  
 for \_, log := range logs {  
 rets = append(rets, \*log)  
 }  
 count, err := isql.OperationLog.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取接口总数失败"))  
 }  
  
 return response.LogListRsp{  
 Total: count,  
 Logs: rets,  
 }, nil  
  
 // 获取  
 // logs, err := isql.OperationLog.List(&r)  
 // if err != nil {  
 // response.Fail(c, nil, "获取操作日志列表失败: "+err.Error())  
 // return  
 // }  
 // return nil, nil  
}  
  
// Delete 删除数据  
func (l OperationLogLogic) Delete(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.OperationLogDeleteReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 for \_, id := range r.OperationLogIds {  
 filter := tools.H{"id": int(id)}  
 if !isql.OperationLog.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("该条记录不存在"))  
 }  
 }  
 // 删除接口  
 err := isql.OperationLog.Delete(r.OperationLogIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("删除该改条记录失败: %s", err.Error()))  
 }  
 return nil, nil  
}  
  
func (l OperationLogLogic) Clean(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 \_, ok := req.(\*request.OperationLogListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 err := isql.OperationLog.Clean()  
 if err != nil {  
 return err, nil  
 }  
 return "操作日志清空完成", nil  
}

package logic  
  
import (  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type passwordPolicyLogic struct{}  
  
var PasswordPolicy = new(passwordPolicyLogic)  
  
// UpdatePolicy 更新密码策略  
func (m \*passwordPolicyLogic) UpdatePolicy(c \*gin.Context, req \*request.PasswordPolicyReq) (interface{}, interface{}) {  
 // 将密码策略保存到数据库  
 err := isql.PasswordPolicy.UpdatePolicy(req)  
 if err != nil {  
 return nil, err  
 }  
 return nil, nil  
}  
  
// GetPolicy 获取密码策略  
func (m \*passwordPolicyLogic) GetPolicy(c \*gin.Context) (interface{}, interface{}) {  
 // 从数据库获取密码策略  
 policy, err := isql.PasswordPolicy.GetPolicy()  
 if err != nil {  
 return nil, err  
 }  
 return policy, nil  
}

package logic  
  
import (  
 "fmt"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/model/response"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
 "github.com/thoas/go-funk"  
)  
  
type ApiLogic struct{}  
  
// Add 添加数据  
func (l ApiLogic) Add(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.ApiAddReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 // 获取当前用户  
 ctxUser, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前登陆用户信息失败"))  
 }  
  
 api := model.Api{  
 Method: r.Method,  
 Path: r.Path,  
 Category: r.Category,  
 Remark: r.Remark,  
 Creator: ctxUser.Username,  
 }  
  
 // 创建接口  
 err = isql.Api.Add(&api)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("创建接口失败: %s", err.Error()))  
 }  
  
 return nil, nil  
}  
  
// List 数据列表  
func (l ApiLogic) List(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.ApiListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 // 获取数据列表  
 apis, err := isql.Api.List(r)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取接口列表失败: %s", err.Error()))  
 }  
  
 rets := make([]model.Api, 0)  
 for \_, api := range apis {  
 rets = append(rets, \*api)  
 }  
 count, err := isql.Api.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取接口总数失败"))  
 }  
  
 return response.ApiListRsp{  
 Total: count,  
 Apis: rets,  
 }, nil  
}  
  
// GetTree 数据树  
func (l ApiLogic) GetTree(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.ApiGetTreeReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 \_ = r  
  
 apis, err := isql.Api.ListAll()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取资源列表失败: " + err.Error()))  
 }  
  
 // 获取所有的分类  
 var categoryList []string  
 for \_, api := range apis {  
 categoryList = append(categoryList, api.Category)  
 }  
 // 获取去重后的分类  
 categoryUniq := funk.UniqString(categoryList)  
  
 apiTree := make([]\*response.ApiTreeRsp, len(categoryUniq))  
  
 for i, category := range categoryUniq {  
 apiTree[i] = &response.ApiTreeRsp{  
 ID: -i,  
 Remark: category,  
 Category: category,  
 Children: nil,  
 }  
 for \_, api := range apis {  
 if category == api.Category {  
 apiTree[i].Children = append(apiTree[i].Children, api)  
 }  
 }  
 }  
  
 return apiTree, nil  
}  
  
// Update 更新数据  
func (l ApiLogic) Update(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.ApiUpdateReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"id": int(r.ID)}  
 if !isql.Api.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("接口不存在"))  
 }  
  
 // 获取当前登陆用户  
 ctxUser, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前登陆用户失败"))  
 }  
  
 oldData := new(model.Api)  
 err = isql.Api.Find(filter, oldData)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
  
 api := model.Api{  
 Model: oldData.Model,  
 Method: r.Method,  
 Path: r.Path,  
 Category: r.Category,  
 Remark: r.Remark,  
 Creator: ctxUser.Username,  
 }  
 err = isql.Api.Update(&api)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("更新接口失败: %s", err.Error()))  
 }  
 return nil, nil  
}  
  
// Delete 删除数据  
func (l ApiLogic) Delete(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.ApiDeleteReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 for \_, id := range r.ApiIds {  
 filter := tools.H{"id": int(id)}  
 if !isql.Api.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("接口不存在"))  
 }  
 }  
 // 删除接口  
 err := isql.Api.Delete(r.ApiIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("删除接口失败: %s", err.Error()))  
 }  
 return nil, nil  
}

package logic  
  
import (  
 "fmt"  
 "strings"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/client/wechat"  
  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type WeComLogic struct {  
}  
  
// 通过企业微信获取部门信息  
func (d \*WeComLogic) SyncWeComDepts(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 // 1.获取所有部门  
 deptSource, err := wechat.GetAllDepts()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("获取企业微信部门列表失败：%s", err.Error()))  
 }  
 depts, err := ConvertDeptData(config.Conf.WeCom.Flag, deptSource)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("转换企业微信部门数据失败：%s", err.Error()))  
 }  
  
 // 2.将远程数据转换成树  
 deptTree := GroupListToTree(fmt.Sprintf("%s\_1", config.Conf.WeCom.Flag), depts)  
  
 // 3.根据树进行创建  
 err = d.addDepts(deptTree.Children)  
  
 return nil, err  
}  
  
// 添加部门  
func (d WeComLogic) addDepts(depts []\*model.Group) error {  
 for \_, dept := range depts {  
 err := d.AddDepts(dept)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("DsyncWeComDepts添加部门失败: %s", err.Error()))  
 }  
 if len(dept.Children) != 0 {  
 err = d.addDepts(dept.Children)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("DsyncWeComDepts添加部门失败: %s", err.Error()))  
 }  
 }  
 }  
 return nil  
}  
  
// AddGroup 添加部门数据  
func (d WeComLogic) AddDepts(group \*model.Group) error {  
 // 判断部门名称是否存在  
 parentGroup := new(model.Group)  
 err := isql.Group.Find(tools.H{"source\_dept\_id": group.SourceDeptParentId}, parentGroup)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("查询父级部门失败：%s", err.Error()))  
 }  
  
 // 此时的 group 已经附带了Build后动态关联好的字段，接下来将一些确定性的其他字段值添加上，就可以创建这个分组了  
 group.Creator = "system"  
 group.GroupType = "cn"  
 group.ParentId = parentGroup.ID  
 group.Source = config.Conf.WeCom.Flag  
 group.GroupDN = fmt.Sprintf("cn=%s,%s", group.GroupName, parentGroup.GroupDN)  
  
 if !isql.Group.Exist(tools.H{"group\_dn": group.GroupDN}) {  
 err = CommonAddGroup(group)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("添加部门: %s, 失败: %s", group.GroupName, err.Error()))  
 }  
 }  
 return nil  
}  
  
// 根据现有数据库同步到的部门信息，开启用户同步  
func (d WeComLogic) SyncWeComUsers(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 // 1.获取企业微信用户列表  
 staffSource, err := wechat.GetAllUsers()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("获取企业微信用户列表失败：%s", err.Error()))  
 }  
 staffs, err := ConvertUserData(config.Conf.WeCom.Flag, staffSource)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("转换企业微信用户数据失败：%s", err.Error()))  
 }  
 // 2.遍历用户，开始写入  
 for \_, staff := range staffs {  
 // 入库  
 err = d.AddUsers(staff)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("SyncWeComUsers写入用户失败：%s", err.Error()))  
 }  
 }  
  
 // 3.获取企业微信已离职用户id列表  
 // 拿到MySQL所有用户数据(来源为 wecom的用户)，远程没有的，则说明被删除了  
 // 如果以后企业微信透出了已离职用户列表的接口，则这里可以进行改进  
 var res []\*model.User  
 users, err := isql.User.ListAll()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取用户列表失败：" + err.Error()))  
 }  
 for \_, user := range users {  
 if user.Source != config.Conf.WeCom.Flag {  
 continue  
 }  
 in := true  
 for \_, staff := range staffs {  
 if user.Username == staff.Username {  
 in = false  
 break  
 }  
 }  
 if in {  
 res = append(res, user)  
 }  
 }  
 // 4.遍历id，开始处理  
 for \_, userTmp := range res {  
 user := new(model.User)  
 err = isql.User.Find(tools.H{"source\_user\_id": userTmp.SourceUserId, "status": 1}, user)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL查询用户失败: " + err.Error()))  
 }  
 // 先从ldap删除用户  
 err = ildap.User.Delete(user.UserDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("在LDAP删除用户失败" + err.Error()))  
 }  
 // 然后更新MySQL中用户状态  
 err = isql.User.ChangeStatus(int(user.ID), 2)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL更新用户状态失败: " + err.Error()))  
 }  
 }  
 return nil, nil  
}  
  
// AddUser 添加用户数据  
func (d WeComLogic) AddUsers(user \*model.User) error {  
 // 根据角色id获取角色  
 roles, err := isql.Role.GetRolesByIds([]uint{2})  
 if err != nil {  
 return tools.NewValidatorError(fmt.Errorf("根据角色ID获取角色信息失败:%s", err.Error()))  
 }  
 user.Creator = "system"  
 user.Roles = roles  
 user.Password = config.Conf.Ldap.UserInitPassword  
 user.Source = config.Conf.WeCom.Flag  
 user.UserDN = fmt.Sprintf("uid=%s,%s", user.Username, config.Conf.Ldap.UserDN)  
  
 // 根据 user\_dn 查询用户,不存在则创建  
 if !isql.User.Exist(tools.H{"user\_dn": user.UserDN}) {  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 var deptTmp string  
 for \_, group := range groups {  
 deptTmp = deptTmp + group.GroupName + ","  
 }  
 user.Departments = strings.TrimRight(deptTmp, ",")  
  
 // 创建用户  
 err = CommonAddUser(user, groups)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("添加用户: %s, 失败: %s", user.Username, err.Error()))  
 }  
 } else {  
 // 此处逻辑未经实际验证，如在使用中有问题，请反馈  
 if config.Conf.WeCom.IsUpdateSyncd {  
 // 先获取用户信息  
 oldData := new(model.User)  
 err = isql.User.Find(tools.H{"user\_dn": user.UserDN}, oldData)  
 if err != nil {  
 return err  
 }  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 var deptTmp string  
 for \_, group := range groups {  
 deptTmp = deptTmp + group.GroupName + ","  
 }  
 user.Model = oldData.Model  
 user.Roles = oldData.Roles  
 user.Creator = oldData.Creator  
 user.Source = oldData.Source  
 user.Password = oldData.Password  
 user.UserDN = oldData.UserDN  
 user.Departments = strings.TrimRight(deptTmp, ",")  
  
 // 用户信息的预置处理  
 if user.Nickname == "" {  
 user.Nickname = oldData.Nickname  
 }  
 if user.GivenName == "" {  
 user.GivenName = user.Nickname  
 }  
 if user.Introduction == "" {  
 user.Introduction = user.Nickname  
 }  
 if user.Mail == "" {  
 user.Mail = oldData.Mail  
 }  
 if user.JobNumber == "" {  
 user.JobNumber = oldData.JobNumber  
 }  
 if user.Departments == "" {  
 user.Departments = oldData.Departments  
 }  
 if user.Position == "" {  
 user.Position = oldData.Position  
 }  
 if user.PostalAddress == "" {  
 user.PostalAddress = oldData.PostalAddress  
 }  
 if user.Mobile == "" {  
 user.Mobile = oldData.Mobile  
 }  
 if err = CommonUpdateUser(oldData, user, tools.StringToSlice(user.DepartmentId, ",")); err != nil {  
 return err  
 }  
 }  
 }  
 return nil  
}

package logic  
  
import (  
 "time"  
  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/public/common"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type ldapMonitorLogic struct{}  
  
var LDAPMonitor = new(ldapMonitorLogic)  
  
// GetStatus 获取LDAP连接状态  
func (m \*ldapMonitorLogic) GetStatus(c \*gin.Context) (interface{}, interface{}) {  
 status, err := isql.LDAPMonitor.GetStatus()  
 if err != nil {  
 return nil, err  
 }  
 return status, nil  
}  
  
// CheckConnection 检查LDAP连接  
func (m \*ldapMonitorLogic) CheckConnection(c \*gin.Context) (interface{}, interface{}) {  
 startTime := time.Now()  
  
 // 获取LDAP连接  
 conn, err := common.GetLDAPConn()  
 if err != nil {  
 return nil, err  
 }  
 defer common.PutLADPConn(conn)  
  
 // 计算响应时间  
 responseTime := time.Since(startTime).Milliseconds()  
  
 // 更新状态  
 status := &request.LDAPMonitorReq{  
 ConnectionStatus: true,  
 ResponseTime: responseTime,  
 LastCheckTime: time.Now().Format("2006-01-02 15:04:05"),  
 ErrorCount: 0,  
 LastError: "",  
 }  
  
 err = isql.LDAPMonitor.UpdateStatus(status)  
 if err != nil {  
 return nil, err  
 }  
  
 return status, nil  
}

package logic  
  
import (  
 "fmt"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "gorm.io/datatypes"  
  
 "github.com/gin-gonic/gin"  
)  
  
type FieldRelationLogic struct{}  
  
// Add 添加数据  
func (l FieldRelationLogic) Add(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.FieldRelationAddReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 if isql.FieldRelation.Exist(tools.H{"flag": r.Flag}) {  
 return nil, tools.NewValidatorError(fmt.Errorf("对应平台的动态字段关系已存在，请勿重复添加"))  
 }  
  
 attr, err := tools.MapToJson(r.Attributes)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("将map转成json失败: %s", err.Error()))  
 }  
  
 frObj := model.FieldRelation{  
 Flag: r.Flag,  
 Attributes: datatypes.JSON(attr),  
 }  
  
 // 创建接口  
 err = isql.FieldRelation.Add(&frObj)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("创建动态字段关系失败: %s", err.Error()))  
 }  
  
 return nil, nil  
}  
  
// List 数据列表  
func (l FieldRelationLogic) List(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 \_, ok := req.(\*request.FieldRelationListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 // 获取数据列表  
 frs, err := isql.FieldRelation.List()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("字段动态关系: %s", err.Error()))  
 }  
  
 return frs, nil  
}  
  
// Update 更新数据  
func (l FieldRelationLogic) Update(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.FieldRelationUpdateReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"flag": r.Flag}  
  
 if !isql.FieldRelation.Exist(filter) {  
 return nil, tools.NewValidatorError(fmt.Errorf("对应平台的动态字段关系不存在"))  
 }  
  
 oldData := new(model.FieldRelation)  
 err := isql.FieldRelation.Find(filter, oldData)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
  
 attr, err := tools.MapToJson(r.Attributes)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("将map转成json失败: %s", err.Error()))  
 }  
  
 frObj := model.FieldRelation{  
 Model: oldData.Model,  
 Flag: r.Flag,  
 Attributes: datatypes.JSON(attr),  
 }  
  
 err = isql.FieldRelation.Update(&frObj)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("更新动态字段关系失败: %s", err.Error()))  
 }  
 return nil, nil  
}  
  
// Delete 删除数据  
func (l FieldRelationLogic) Delete(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.FieldRelationDeleteReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 for \_, id := range r.FieldRelationIds {  
 filter := tools.H{"id": int(id)}  
 if !isql.FieldRelation.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("动态字段关系不存在"))  
 }  
 }  
 // 删除  
 err := isql.FieldRelation.Delete(r.FieldRelationIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("删除动态字段关系失败: %s", err.Error()))  
 }  
 return nil, nil  
}

package logic  
  
import (  
 "fmt"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type SqlLogic struct{}  
  
// SyncSqlUsers 同步sql的用户信息到ldap  
func (d \*SqlLogic) SyncSqlUsers(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.SyncSqlUserReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 // 1.获取所有用户  
 for \_, id := range r.UserIds {  
 filter := tools.H{"id": int(id)}  
 if !isql.User.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("有用户不存在"))  
 }  
 }  
 users, err := isql.User.GetUserByIds(r.UserIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取用户信息失败: " + err.Error()))  
 }  
 // 2.再将用户添加到ldap  
 for \_, user := range users {  
 err = ildap.User.Add(&user)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("SyncUser向LDAP同步用户失败：" + err.Error()))  
 }  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 for \_, group := range groups {  
 //根据选择的部门，添加到部门内  
 err = ildap.Group.AddUserToGroup(group.GroupDN, user.UserDN)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("向Ldap添加用户到分组关系失败：" + err.Error()))  
 }  
 }  
 err = isql.User.ChangeSyncState(int(user.ID), 1)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("用户同步完毕之后更新状态失败：" + err.Error()))  
 }  
 }  
  
 return nil, nil  
}  
  
// SyncSqlGroups 同步sql中的分组信息到ldap  
func (d \*SqlLogic) SyncSqlGroups(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.SyncSqlGrooupsReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 // 1.获取所有分组  
 for \_, id := range r.GroupIds {  
 filter := tools.H{"id": int(id)}  
 if !isql.Group.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("有分组不存在"))  
 }  
 }  
 groups, err := isql.Group.GetGroupByIds(r.GroupIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组信息失败: " + err.Error()))  
 }  
 // 2.再将分组添加到ldap  
 for \_, group := range groups {  
 err = ildap.Group.Add(group)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("SyncUser向LDAP同步分组失败：" + err.Error()))  
 }  
 if len(group.Users) > 0 {  
 for \_, user := range group.Users {  
 if user.UserDN == config.Conf.Ldap.AdminDN {  
 continue  
 }  
 err = ildap.Group.AddUserToGroup(group.GroupDN, user.UserDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("同步分组之后处理分组内的用户失败：" + err.Error()))  
 }  
 }  
 }  
 err = isql.Group.ChangeSyncState(int(group.ID), 1)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("分组同步完毕之后更新状态失败：" + err.Error()))  
 }  
 }  
  
 return nil, nil  
}  
  
// SearchGroupDiff 检索未同步到ldap中的分组  
func SearchGroupDiff() (err error) {  
 // 获取sql中的数据  
 var sqlGroupList []\*model.Group  
 sqlGroupList, err = isql.Group.ListAll()  
 if err != nil {  
 return err  
 }  
 // 获取ldap中的数据  
 var ldapGroupList []\*model.Group  
 ldapGroupList, err = ildap.Group.ListGroupDN()  
 if err != nil {  
 return err  
 }  
 // 比对两个系统中的数据  
 groups := diffGroup(sqlGroupList, ldapGroupList)  
 for \_, group := range groups {  
 if group.GroupDN == config.Conf.Ldap.BaseDN {  
 continue  
 }  
 err = isql.Group.ChangeSyncState(int(group.ID), 2)  
 }  
 return  
}  
  
// SearchUserDiff 检索未同步到ldap中的用户  
func SearchUserDiff() (err error) {  
 // 获取sql中的数据  
 var sqlUserList []\*model.User  
 sqlUserList, err = isql.User.ListAll()  
 if err != nil {  
 return err  
 }  
 // 获取ldap中的数据  
 var ldapUserList []\*model.User  
 ldapUserList, err = ildap.User.ListUserDN()  
 if err != nil {  
 return err  
 }  
 // 比对两个系统中的数据  
 users := diffUser(sqlUserList, ldapUserList)  
 for \_, user := range users {  
 if user.UserDN == config.Conf.Ldap.AdminDN {  
 continue  
 }  
 err = isql.User.ChangeSyncState(int(user.ID), 2)  
 }  
 return  
}  
  
// diffGroup 比较出sql中有但ldap中没有的group列表  
func diffGroup(sqlGroup, ldapGroup []\*model.Group) (rst []\*model.Group) {  
 var tmp = make(map[string]struct{}, 0)  
  
 for \_, v := range ldapGroup {  
 tmp[v.GroupDN] = struct{}{}  
 }  
  
 for \_, v := range sqlGroup {  
 if \_, ok := tmp[v.GroupDN]; !ok {  
 rst = append(rst, v)  
 }  
 }  
 return  
}  
  
// diffUser 比较出sql中有但ldap中没有的user列表  
func diffUser(sqlUser, ldapUser []\*model.User) (rst []\*model.User) {  
 var tmp = make(map[string]struct{}, len(sqlUser))  
  
 for \_, v := range ldapUser {  
 tmp[v.UserDN] = struct{}{}  
 }  
  
 for \_, v := range sqlUser {  
 if \_, ok := tmp[v.UserDN]; !ok {  
 rst = append(rst, v)  
 }  
 }  
 return  
}

package logic  
  
import (  
 "fmt"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/model/response"  
 "github.com/eryajf/go-ldap-admin/public/common"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
 "github.com/thoas/go-funk"  
)  
  
type RoleLogic struct{}  
  
// Add 添加数据  
func (l RoleLogic) Add(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.RoleAddReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 if isql.Role.Exist(tools.H{"name": r.Name}) {  
 return nil, tools.NewValidatorError(fmt.Errorf("该角色名已存在"))  
 }  
  
 // 获取当前用户最高角色等级  
 minSort, ctxUser, err := isql.User.GetCurrentUserMinRoleSort(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前用户最高角色等级失败: %s", err.Error()))  
 }  
 if minSort != 1 {  
 return nil, tools.NewValidatorError(fmt.Errorf("当前用户没有权限更新角色"))  
 }  
 // 用户不能创建比自己等级高或相同等级的角色  
 if minSort >= r.Sort {  
 return nil, tools.NewValidatorError(fmt.Errorf("不能创建比自己等级高或相同等级的角色"))  
 }  
  
 role := model.Role{  
 Name: r.Name,  
 Keyword: r.Keyword,  
 Remark: r.Remark,  
 Status: r.Status,  
 Sort: r.Sort,  
 Creator: ctxUser.Username,  
 }  
  
 // 创建角色  
 err = isql.Role.Add(&role)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("创建角色失败: %s", err.Error()))  
 }  
 return nil, nil  
}  
  
// List 数据列表  
func (l RoleLogic) List(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.RoleListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 // 获取数据列表  
 roles, err := isql.Role.List(r)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取菜单列表失败: %s", err.Error()))  
 }  
  
 count, err := isql.Role.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取接口总数失败"))  
 }  
  
 rets := make([]model.Role, 0)  
 for \_, role := range roles {  
 rets = append(rets, \*role)  
 }  
  
 return response.RoleListRsp{  
 Total: count,  
 Roles: rets,  
 }, nil  
}  
  
// Update 更新数据  
func (l RoleLogic) Update(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.RoleUpdateReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"id": r.ID}  
 if !isql.Role.Exist(filter) {  
 return nil, tools.NewValidatorError(fmt.Errorf("该角色名不已存在"))  
 }  
  
 // 当前用户角色排序最小值（最高等级角色）以及当前用户  
 minSort, ctxUser, err := isql.User.GetCurrentUserMinRoleSort(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前用户最高角色等级失败: %s", err.Error()))  
 }  
  
 if minSort != 1 {  
 return nil, tools.NewValidatorError(fmt.Errorf("当前用户没有权限更新角色"))  
 }  
  
 // 不能更新比自己角色等级高或相等的角色  
 // 根据path中的角色ID获取该角色信息  
 roles, \_ := isql.Role.GetRolesByIds([]uint{r.ID})  
 if len(roles) == 0 {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取角色信息失败: %s", err.Error()))  
 }  
  
 if minSort >= roles[0].Sort {  
 return nil, tools.NewValidatorError(fmt.Errorf("不能更新比自己角色等级高或相等的角色"))  
 }  
  
 // 不能把角色等级更新得比当前用户的等级高  
 if minSort >= r.Sort {  
 return nil, tools.NewValidatorError(fmt.Errorf("不能把角色等级更新得比当前用户的等级高或相同"))  
 }  
 oldData := new(model.Role)  
 err = isql.Role.Find(filter, oldData)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
 role := model.Role{  
 Model: oldData.Model,  
 Name: r.Name,  
 Keyword: r.Keyword,  
 Remark: r.Remark,  
 Status: r.Status,  
 Sort: r.Sort,  
 Creator: ctxUser.Username,  
 }  
  
 // 更新角色  
 err = isql.Role.Update(&role)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("更新角色失败: %s", err.Error()))  
 }  
  
 // 如果更新成功，且更新了角色的keyword, 则更新casbin中policy  
 if r.Keyword != roles[0].Keyword {  
 // 获取policy  
 rolePolicies := common.CasbinEnforcer.GetFilteredPolicy(0, roles[0].Keyword)  
 if len(rolePolicies) == 0 {  
 return  
 }  
 rolePoliciesCopy := make([][]string, 0)  
 // 替换keyword  
 for \_, policy := range rolePolicies {  
 policyCopy := make([]string, len(policy))  
 copy(policyCopy, policy)  
 rolePoliciesCopy = append(rolePoliciesCopy, policyCopy)  
 policy[0] = r.Keyword  
 }  
  
 //gormadapter未实现UpdatePolicies方法，等gorm更新---  
 //isUpdated, \_ := common.CasbinEnforcer.UpdatePolicies(rolePoliciesCopy, rolePolicies)  
 //if !isUpdated {  
 // response.Fail(c, nil, "更新角色成功，但角色关键字关联的权限接口更新失败！")  
 // return  
 //}  
  
 // 这里需要先新增再删除（先删除再增加会出错）  
 isAdded, \_ := common.CasbinEnforcer.AddPolicies(rolePolicies)  
 if !isAdded {  
 return nil, tools.NewOperationError(fmt.Errorf("更新角色成功，但角色关键字关联的权限接口更新失败"))  
 }  
 isRemoved, \_ := common.CasbinEnforcer.RemovePolicies(rolePoliciesCopy)  
 if !isRemoved {  
 return nil, tools.NewOperationError(fmt.Errorf("更新角色成功，但角色关键字关联的权限接口更新失败"))  
 }  
 err := common.CasbinEnforcer.LoadPolicy()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("更新角色成功，但角色关键字关联角色的权限接口策略加载失败"))  
 }  
  
 }  
  
 // 更新角色成功处理用户信息缓存有两种做法:（这里使用第二种方法，因为一个角色下用户数量可能很多，第二种方法可以分散数据库压力）  
 // 1.可以帮助用户更新拥有该角色的用户信息缓存,使用下面方法  
 // err = ur.UpdateUserInfoCacheByRoleId(uint(roleId))  
 // 2.直接清理缓存，让活跃的用户自己重新缓存最新用户信息  
 isql.User.ClearUserInfoCache()  
  
 return nil, nil  
}  
  
// Delete 删除数据  
func (l RoleLogic) Delete(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.RoleDeleteReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 // 获取当前登陆用户最高等级角色  
 minSort, \_, err := isql.User.GetCurrentUserMinRoleSort(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前用户最高角色等级失败: %s", err.Error()))  
 }  
  
 // 获取角色信息  
 roles, \_ := isql.Role.GetRolesByIds(r.RoleIds)  
 if len(roles) == 0 {  
 return nil, tools.NewMySqlError(fmt.Errorf("未能获取到角色信息"))  
 }  
  
 // 不能删除比自己角色等级高或相等的角色  
 for \_, role := range roles {  
 if minSort >= role.Sort {  
 return nil, tools.NewValidatorError(fmt.Errorf("不能删除比自己角色等级高或相等的角色"))  
 }  
 }  
  
 // 删除角色  
 err = isql.Role.Delete(r.RoleIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("删除角色失败: %s", err.Error()))  
 }  
  
 // 删除角色成功直接清理缓存，让活跃的用户自己重新缓存最新用户信息  
 isql.User.ClearUserInfoCache()  
 return nil, nil  
}  
  
// GetMenuList 获取角色菜单列表  
func (l RoleLogic) GetMenuList(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.RoleGetMenuListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 menus, err := isql.Role.GetRoleMenusById(r.RoleID)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取角色的权限菜单失败: " + err.Error()))  
 }  
 return menus, nil  
}  
  
// GetApiList 获取角色接口列表  
func (l RoleLogic) GetApiList(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.RoleGetApiListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 role := new(model.Role)  
 err := isql.Role.Find(tools.H{"id": r.RoleID}, role)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取资源失败: " + err.Error()))  
 }  
  
 policies := common.CasbinEnforcer.GetFilteredPolicy(0, role.Keyword)  
  
 apis, err := isql.Api.ListAll()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取资源列表失败: " + err.Error()))  
 }  
 accessApis := make([]\*model.Api, 0)  
  
 for \_, policy := range policies {  
 path := policy[1]  
 method := policy[2]  
 for \_, api := range apis {  
 if path == api.Path && method == api.Method {  
 accessApis = append(accessApis, api)  
 break  
 }  
 }  
 }  
  
 return accessApis, nil  
}  
  
// UpdateMenus 更新角色菜单  
func (l RoleLogic) UpdateMenus(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.RoleUpdateMenusReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 roles, \_ := isql.Role.GetRolesByIds([]uint{r.RoleID})  
 if len(roles) == 0 {  
 return nil, tools.NewMySqlError(fmt.Errorf("未获取到角色信息"))  
 }  
  
 // 当前用户角色排序最小值（最高等级角色）以及当前用户  
 minSort, ctxUser, err := isql.User.GetCurrentUserMinRoleSort(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前用户最高角色等级失败: %s", err.Error()))  
 }  
  
 // (非管理员)不能更新比自己角色等级高或相等角色的权限菜单  
 if minSort != 1 {  
 if minSort >= roles[0].Sort {  
 return nil, tools.NewValidatorError(fmt.Errorf("不能更新比自己角色等级高或相等角色的权限菜单"))  
 }  
 }  
  
 // 获取当前用户所拥有的权限菜单  
 ctxUserMenus, err := isql.Menu.GetUserMenusByUserId(ctxUser.ID)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前用户的可访问菜单列表失败: " + err.Error()))  
 }  
  
 // 获取当前用户所拥有的权限菜单ID  
 ctxUserMenusIds := make([]uint, 0)  
 for \_, menu := range ctxUserMenus {  
 ctxUserMenusIds = append(ctxUserMenusIds, menu.ID)  
 }  
  
 // 用户需要修改的菜单集合  
 reqMenus := make([]\*model.Menu, 0)  
  
 // (非管理员)不能把角色的权限菜单设置的比当前用户所拥有的权限菜单多  
 if minSort != 1 {  
 for \_, id := range r.MenuIds {  
 if !funk.Contains(ctxUserMenusIds, id) {  
 return nil, tools.NewValidatorError(fmt.Errorf("无权设置ID为%d的菜单", id))  
 }  
 }  
  
 for \_, id := range r.MenuIds {  
 for \_, menu := range ctxUserMenus {  
 if id == menu.ID {  
 reqMenus = append(reqMenus, menu)  
 break  
 }  
 }  
 }  
 } else {  
 // 管理员随意设置  
 // 根据menuIds查询查询菜单  
 menus, err := isql.Menu.List()  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("获取菜单列表失败: " + err.Error()))  
 }  
 for \_, menuId := range r.MenuIds {  
 for \_, menu := range menus {  
 if menuId == menu.ID {  
 reqMenus = append(reqMenus, menu)  
 }  
 }  
 }  
 }  
  
 roles[0].Menus = reqMenus  
  
 err = isql.Role.UpdateRoleMenus(roles[0])  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("更新角色的权限菜单失败: " + err.Error()))  
 }  
  
 return nil, nil  
}  
  
// UpdateApis 更新角色接口  
func (l RoleLogic) UpdateApis(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.RoleUpdateApisReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 // 根据path中的角色ID获取该角色信息  
 roles, \_ := isql.Role.GetRolesByIds([]uint{r.RoleID})  
 if len(roles) == 0 {  
 return nil, tools.NewMySqlError(fmt.Errorf("未获取到角色信息"))  
 }  
  
 // 当前用户角色排序最小值（最高等级角色）以及当前用户  
 minSort, ctxUser, err := isql.User.GetCurrentUserMinRoleSort(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前用户最高角色等级失败: %s", err.Error()))  
 }  
  
 // (非管理员)不能更新比自己角色等级高或相等角色的权限菜单  
 if minSort != 1 {  
 if minSort >= roles[0].Sort {  
 return nil, tools.NewValidatorError(fmt.Errorf("不能更新比自己角色等级高或相等角色的权限菜单"))  
 }  
 }  
  
 // 获取当前用户所拥有的权限接口  
 ctxRoles := ctxUser.Roles  
 ctxRolesPolicies := make([][]string, 0)  
 for \_, role := range ctxRoles {  
 policy := common.CasbinEnforcer.GetFilteredPolicy(0, role.Keyword)  
 ctxRolesPolicies = append(ctxRolesPolicies, policy...)  
 }  
 // 得到path中的角色ID对应角色能够设置的权限接口集合  
 for \_, policy := range ctxRolesPolicies {  
 policy[0] = roles[0].Keyword  
 }  
  
 // 根据apiID获取接口详情  
 apis, err := isql.Api.GetApisById(r.ApiIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("根据接口ID获取接口信息失败"))  
 }  
 // 生成前端想要设置的角色policies  
 reqRolePolicies := make([][]string, 0)  
 for \_, api := range apis {  
 reqRolePolicies = append(reqRolePolicies, []string{  
 roles[0].Keyword, api.Path, api.Method,  
 })  
 }  
  
 // (非管理员)不能把角色的权限接口设置的比当前用户所拥有的权限接口多  
 if minSort != 1 {  
 for \_, reqPolicy := range reqRolePolicies {  
 if !funk.Contains(ctxRolesPolicies, reqPolicy) {  
 return nil, tools.NewValidatorError(fmt.Errorf("无权设置路径为%s,请求方式为%s的接口", reqPolicy[1], reqPolicy[2]))  
 }  
 }  
 }  
  
 // 更新角色的权限接口  
 err = isql.Role.UpdateRoleApis(roles[0].Keyword, reqRolePolicies)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("更新角色的权限接口失败"))  
 }  
 return nil, nil  
}

package logic  
  
import (  
 "fmt"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
)  
  
type MenuLogic struct{}  
  
// Add 添加数据  
func (l MenuLogic) Add(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.MenuAddReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 if isql.Menu.Exist(tools.H{"name": r.Name}) {  
 return nil, tools.NewMySqlError(fmt.Errorf("菜单名称已存在"))  
  
 }  
  
 // 获取当前用户  
 ctxUser, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前登陆用户信息失败"))  
 }  
  
 menu := model.Menu{  
 Name: r.Name,  
 Title: r.Title,  
 Icon: r.Icon,  
 Path: r.Path,  
 Redirect: r.Redirect,  
 Component: r.Component,  
 Sort: r.Sort,  
 Status: r.Status,  
 Hidden: r.Hidden,  
 NoCache: r.NoCache,  
 AlwaysShow: r.AlwaysShow,  
 Breadcrumb: r.Breadcrumb,  
 ActiveMenu: r.ActiveMenu,  
 ParentId: r.ParentId,  
 Creator: ctxUser.Username,  
 }  
  
 err = isql.Menu.Add(&menu)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("创建记录失败: %s", err.Error()))  
 }  
  
 return nil, nil  
}  
  
// // List 数据列表  
// func (l MenuLogic) List(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
// \_, ok := req.(\*request.MenuListReq)  
// if !ok {  
// return nil, ReqAssertErr  
// }  
// \_ = c  
  
// menus, err := isql.Menu.List()  
// if err != nil {  
// return nil, tools.NewMySqlError(fmt.Errorf("获取资源列表失败: %s", err.Error()))  
// }  
  
// rets := make([]model.Menu, 0)  
// for \_, menu := range menus {  
// rets = append(rets, \*menu)  
// }  
// count, err := isql.Menu.Count()  
// if err != nil {  
// return nil, tools.NewMySqlError(fmt.Errorf("获取资源总数失败"))  
// }  
  
// return response.MenuListRsp{  
// Total: count,  
// Menus: rets,  
// }, nil  
// }  
  
// Update 更新数据  
func (l MenuLogic) Update(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.MenuUpdateReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"id": int(r.ID)}  
 if !isql.Menu.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("该ID对应的记录不存在"))  
 }  
  
 // 获取当前登陆用户  
 ctxUser, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前登陆用户失败"))  
 }  
  
 oldData := new(model.Menu)  
 err = isql.Menu.Find(filter, oldData)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取记录失败: %s", err.Error()))  
 }  
  
 menu := model.Menu{  
 Model: oldData.Model,  
 Name: r.Name,  
 Title: r.Title,  
 Icon: r.Icon,  
 Path: r.Path,  
 Redirect: r.Redirect,  
 Component: r.Component,  
 Sort: r.Sort,  
 Status: r.Status,  
 Hidden: r.Hidden,  
 NoCache: r.NoCache,  
 AlwaysShow: r.AlwaysShow,  
 Breadcrumb: r.Breadcrumb,  
 ActiveMenu: r.ActiveMenu,  
 ParentId: r.ParentId,  
 Creator: ctxUser.Username,  
 }  
  
 err = isql.Menu.Update(&menu)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("更新记录失败: %s", err.Error()))  
 }  
  
 return nil, nil  
}  
  
// Delete 删除数据  
func (l MenuLogic) Delete(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.MenuDeleteReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 for \_, id := range r.MenuIds {  
 filter := tools.H{"id": int(id)}  
 if !isql.Menu.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("该ID对应的记录不存在"))  
 }  
 }  
  
 // 删除接口  
 err := isql.Menu.Delete(r.MenuIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("删除接口失败: %s", err.Error()))  
 }  
 return nil, nil  
}  
  
// GetTree 数据树  
func (l MenuLogic) GetTree(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 \_, ok := req.(\*request.MenuGetTreeReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 menus, err := isql.Menu.List()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取资源列表失败: " + err.Error()))  
 }  
  
 tree := isql.GenMenuTree(0, menus)  
  
 return tree, nil  
}  
  
// GetAccessTree 获取用户菜单树  
func (l MenuLogic) GetAccessTree(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.MenuGetAccessTreeReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 // 校验  
 filter := tools.H{"id": r.ID}  
 if !isql.User.Exist(filter) {  
 return nil, tools.NewValidatorError(fmt.Errorf("该用户不存在"))  
 }  
 user := new(model.User)  
 err := isql.User.Find(filter, user)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL查询用户失败: " + err.Error()))  
 }  
 var roleIds []uint  
 for \_, role := range user.Roles {  
 roleIds = append(roleIds, role.ID)  
 }  
 menus, err := isql.Menu.ListUserMenus(roleIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取资源列表失败: " + err.Error()))  
 }  
  
 tree := isql.GenMenuTree(0, menus)  
  
 return tree, nil  
}

package logic  
  
import (  
 "encoding/csv"  
 "fmt"  
 "os"  
 "time"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type userImportLogic struct{}  
  
var UserImport = new(userImportLogic)  
  
// ImportUsers 批量导入用户  
func (m \*userImportLogic) ImportUsers(c \*gin.Context, req \*request.UserImportReq) (interface{}, interface{}) {  
 // 批量导入用户到LDAP  
 for \_, userItem := range req.Users {  
 // 创建用户对象  
 user := &model.User{  
 Username: userItem.Username,  
 Nickname: userItem.Nickname,  
 Mail: userItem.Email,  
 Mobile: userItem.Phone,  
 Departments: userItem.Department,  
 Position: userItem.Position,  
 Introduction: userItem.Description,  
 Status: 1, // 默认在职  
 Source: "platform",  
 DepartmentId: "0", // 默认部门ID  
 UserDN: fmt.Sprintf("uid=%s,%s", userItem.Username, config.Conf.Ldap.UserDN),  
 }  
  
 // 生成随机密码  
 password := tools.RandomString(12)  
 user.Password = tools.NewGenPasswd(password)  
  
 // 保存到数据库  
 err := isql.User.Add(user)  
 if err != nil {  
 return nil, err  
 }  
  
 // 同步到LDAP  
 err = ildap.User.Add(user)  
 if err != nil {  
 // 如果LDAP同步失败，删除数据库中的用户  
 isql.User.Delete([]uint{user.ID})  
 return nil, err  
 }  
  
 // 更新同步状态  
 user.SyncState = 1  
 isql.User.Update(user)  
 }  
  
 return nil, nil  
}  
  
// ExportUsers 导出用户  
func (m \*userImportLogic) ExportUsers(c \*gin.Context) (interface{}, interface{}) {  
 // 获取所有用户  
 users, err := isql.User.ListAll()  
 if err != nil {  
 return nil, err  
 }  
  
 // 创建CSV文件  
 filename := fmt.Sprintf("users\_%s.csv", time.Now().Format("20060102150405"))  
 file, err := os.Create(filename)  
 if err != nil {  
 return nil, err  
 }  
 defer file.Close()  
  
 // 创建CSV写入器  
 writer := csv.NewWriter(file)  
 defer writer.Flush()  
  
 // 写入表头  
 headers := []string{"用户名", "昵称", "邮箱", "手机号", "部门", "职位", "描述"}  
 if err := writer.Write(headers); err != nil {  
 return nil, err  
 }  
  
 // 写入用户数据  
 for \_, user := range users {  
 record := []string{  
 user.Username,  
 user.Nickname,  
 user.Mail,  
 user.Mobile,  
 user.Departments,  
 user.Position,  
 user.Introduction,  
 }  
 if err := writer.Write(record); err != nil {  
 return nil, err  
 }  
 }  
  
 return filename, nil  
}

package logic  
  
import (  
 "fmt"  
 "strings"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/client/feishu"  
  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type FeiShuLogic struct {  
}  
  
// 通过飞书获取部门信息  
func (d \*FeiShuLogic) SyncFeiShuDepts(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 // 1.获取所有部门  
 deptSource, err := feishu.GetAllDepts()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("获取飞书部门列表失败：%s", err.Error()))  
 }  
 depts, err := ConvertDeptData(config.Conf.FeiShu.Flag, deptSource)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("转换飞书部门数据失败：%s", err.Error()))  
 }  
  
 // 2.将远程数据转换成树  
 deptTree := GroupListToTree(fmt.Sprintf("%s\_0", config.Conf.FeiShu.Flag), depts)  
  
 // 3.根据树进行创建  
 err = d.addDepts(deptTree.Children)  
  
 return nil, err  
}  
  
// 添加部门  
func (d FeiShuLogic) addDepts(depts []\*model.Group) error {  
 for \_, dept := range depts {  
 err := d.AddDepts(dept)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("DsyncFeiShuDepts添加部门失败: %s", err.Error()))  
 }  
 if len(dept.Children) != 0 {  
 err = d.addDepts(dept.Children)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("DsyncFeiShuDepts添加部门失败: %s", err.Error()))  
 }  
 }  
 }  
 return nil  
}  
  
// AddGroup 添加部门数据  
func (d FeiShuLogic) AddDepts(group \*model.Group) error {  
 // 查询当前分组父ID在MySQL中的数据信息  
 parentGroup := new(model.Group)  
 err := isql.Group.Find(tools.H{"source\_dept\_id": group.SourceDeptParentId}, parentGroup)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("查询父级部门失败：%s", err.Error()))  
 }  
  
 // 此时的 group 已经附带了Build后动态关联好的字段，接下来将一些确定性的其他字段值添加上，就可以创建这个分组了  
 group.Creator = "system"  
 group.GroupType = "cn"  
 group.ParentId = parentGroup.ID  
 group.Source = config.Conf.FeiShu.Flag  
 group.GroupDN = fmt.Sprintf("cn=%s,%s", group.GroupName, parentGroup.GroupDN)  
  
 if !isql.Group.Exist(tools.H{"group\_dn": group.GroupDN}) {  
 err = CommonAddGroup(group)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("添加部门: %s, 失败: %s", group.GroupName, err.Error()))  
 }  
 }  
 return nil  
}  
  
// 根据现有数据库同步到的部门信息，开启用户同步  
func (d FeiShuLogic) SyncFeiShuUsers(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 // 1.获取飞书用户列表  
 staffSource, err := feishu.GetAllUsers()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("获取飞书用户列表失败：%s", err.Error()))  
 }  
 staffs, err := ConvertUserData(config.Conf.FeiShu.Flag, staffSource)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("转换飞书用户数据失败：%s", err.Error()))  
 }  
 // 2.遍历用户，开始写入  
 for \_, staff := range staffs {  
 // 入库  
 err = d.AddUsers(staff)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("SyncFeiShuUsers写入用户失败：%s", err.Error()))  
 }  
 }  
  
 // 3.获取飞书已离职用户id列表  
 userIds, err := feishu.GetLeaveUserIds()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("SyncFeiShuUsers获取飞书离职用户列表失败：%s", err.Error()))  
 }  
 // 4.遍历id，开始处理  
 for \_, uid := range userIds {  
 if isql.User.Exist(  
 tools.H{  
 "status": 1, //只处理1在职的  
 "source\_union\_id": fmt.Sprintf("%s\_%s", config.Conf.FeiShu.Flag, uid),  
 }) {  
 user := new(model.User)  
 err = isql.User.Find(tools.H{"source\_union\_id": fmt.Sprintf("%s\_%s", config.Conf.FeiShu.Flag, uid)}, user)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL查询用户失败: " + err.Error()))  
 }  
 // 先从ldap删除用户  
 err = ildap.User.Delete(user.UserDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("在LDAP删除用户失败" + err.Error()))  
 }  
 // 然后更新MySQL中用户状态  
 err = isql.User.ChangeStatus(int(user.ID), 2)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL更新用户状态失败: " + err.Error()))  
 }  
 }  
 }  
  
 return nil, nil  
}  
  
// AddUser 添加用户数据  
func (d FeiShuLogic) AddUsers(user \*model.User) error {  
 // 根据角色id获取角色  
 roles, err := isql.Role.GetRolesByIds([]uint{2})  
 if err != nil {  
 return tools.NewValidatorError(fmt.Errorf("根据角色ID获取角色信息失败:%s", err.Error()))  
 }  
 user.Roles = roles  
 user.Creator = "system"  
 user.Source = config.Conf.FeiShu.Flag  
 user.Password = config.Conf.Ldap.UserInitPassword  
 user.UserDN = fmt.Sprintf("uid=%s,%s", user.Username, config.Conf.Ldap.UserDN)  
  
 // 根据 user\_dn 查询用户,不存在则创建  
 if !isql.User.Exist(tools.H{"user\_dn": user.UserDN}) {  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 var deptTmp string  
 for \_, group := range groups {  
 deptTmp = deptTmp + group.GroupName + ","  
 }  
 user.Departments = strings.TrimRight(deptTmp, ",")  
  
 // 添加用户  
 err = CommonAddUser(user, groups)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("添加用户: %s, 失败: %s", user.Username, err.Error()))  
 }  
 } else {  
 // 此处逻辑未经实际验证，如在使用中有问题，请反馈  
 if config.Conf.FeiShu.IsUpdateSyncd {  
 // 先获取用户信息  
 oldData := new(model.User)  
 err = isql.User.Find(tools.H{"user\_dn": user.UserDN}, oldData)  
 if err != nil {  
 return err  
 }  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 var deptTmp string  
 for \_, group := range groups {  
 deptTmp = deptTmp + group.GroupName + ","  
 }  
 user.Model = oldData.Model  
 user.Roles = oldData.Roles  
 user.Creator = oldData.Creator  
 user.Source = oldData.Source  
 user.Password = oldData.Password  
 user.UserDN = oldData.UserDN  
 user.Departments = strings.TrimRight(deptTmp, ",")  
  
 // 用户信息的预置处理  
 if user.Nickname == "" {  
 user.Nickname = oldData.Nickname  
 }  
 if user.GivenName == "" {  
 user.GivenName = user.Nickname  
 }  
 if user.Introduction == "" {  
 user.Introduction = user.Nickname  
 }  
 if user.Mail == "" {  
 user.Mail = oldData.Mail  
 }  
 if user.JobNumber == "" {  
 user.JobNumber = oldData.JobNumber  
 }  
 if user.Departments == "" {  
 user.Departments = oldData.Departments  
 }  
 if user.Position == "" {  
 user.Position = oldData.Position  
 }  
 if user.PostalAddress == "" {  
 user.PostalAddress = oldData.PostalAddress  
 }  
 if user.Mobile == "" {  
 user.Mobile = oldData.Mobile  
 }  
 if err = CommonUpdateUser(oldData, user, tools.StringToSlice(user.DepartmentId, ",")); err != nil {  
 return err  
 }  
 }  
 }  
 return nil  
}

package logic  
  
import (  
 "fmt"  
 "strings"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/client/openldap"  
  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type OpenLdapLogic struct {  
}  
  
//通过ldap获取部门信息  
func (d \*OpenLdapLogic) SyncOpenLdapDepts(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 // 1.获取所有部门  
 depts, err := openldap.GetAllDepts()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("获取ldap部门列表失败：%s", err.Error()))  
 }  
 groups := make([]\*model.Group, 0)  
 for \_, dept := range depts {  
 groups = append(groups, &model.Group{  
 GroupName: dept.Name,  
 Remark: dept.Remark,  
 SourceDeptId: dept.Id,  
 SourceDeptParentId: dept.ParentId,  
 GroupDN: dept.DN,  
 })  
 }  
 // 2.将远程数据转换成树  
 deptTree := GroupListToTree("0", groups)  
  
 // 3.根据树进行创建  
 err = d.addDepts(deptTree.Children)  
  
 return nil, err  
}  
  
// 添加部门  
func (d OpenLdapLogic) addDepts(depts []\*model.Group) error {  
 for \_, dept := range depts {  
 err := d.AddDepts(dept)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("DsyncOpenLdapDepts添加部门失败: %s", err.Error()))  
 }  
 if len(dept.Children) != 0 {  
 err = d.addDepts(dept.Children)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("DsyncOpenLdapDepts添加部门失败: %s", err.Error()))  
 }  
 }  
 }  
 return nil  
}  
  
// AddGroup 添加部门数据  
func (d OpenLdapLogic) AddDepts(group \*model.Group) error {  
 // 判断部门名称是否存在,此处使用ldap中的唯一值dn,以免出现数据同步不全的问题  
 if !isql.Group.Exist(tools.H{"group\_dn": group.GroupDN}) {  
 // 此时的 group 已经附带了Build后动态关联好的字段，接下来将一些确定性的其他字段值添加上，就可以创建这个分组了  
 group.Creator = "system"  
 group.GroupType = strings.Split(strings.Split(group.GroupDN, ",")[0], "=")[0]  
 parentid, err := d.getParentGroupID(group)  
 if err != nil {  
 return err  
 }  
 group.ParentId = parentid  
 group.Source = "openldap"  
 err = isql.Group.Add(group)  
 if err != nil {  
 return err  
 }  
 }  
 return nil  
  
}  
  
// AddGroup 添加部门数据  
func (d OpenLdapLogic) getParentGroupID(group \*model.Group) (id uint, err error) {  
 switch group.SourceDeptParentId {  
 case "dingtalkroot":  
 group.SourceDeptParentId = "dingtalk\_1"  
 case "feishuroot":  
 group.SourceDeptParentId = "feishu\_0"  
 case "wecomroot":  
 group.SourceDeptParentId = "wecom\_1"  
 }  
 parentGroup := new(model.Group)  
 err = isql.Group.Find(tools.H{"source\_dept\_id": group.SourceDeptParentId}, parentGroup)  
 if err != nil {  
 return id, tools.NewMySqlError(fmt.Errorf("查询父级部门失败：%s,%s", err.Error(), group.GroupName))  
 }  
 return parentGroup.ID, nil  
}  
  
//根据现有数据库同步到的部门信息，开启用户同步  
func (d OpenLdapLogic) SyncOpenLdapUsers(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 // 1.获取ldap用户列表  
 staffs, err := openldap.GetAllUsers()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("获取ldap用户列表失败：%s", err.Error()))  
 }  
 // 2.遍历用户，开始写入  
 for \_, staff := range staffs {  
 groupIds, err := isql.Group.DeptIdsToGroupIds(staff.DepartmentIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("将部门ids转换为内部部门id失败：%s", err.Error()))  
 }  
 // 根据角色id获取角色  
 roles, err := isql.Role.GetRolesByIds([]uint{2})  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("根据角色ID获取角色信息失败:%s", err.Error()))  
 }  
 // 入库  
 err = d.AddUsers(&model.User{  
 Username: staff.Name,  
 Nickname: staff.DisplayName,  
 GivenName: staff.GivenName,  
 Mail: staff.Mail,  
 JobNumber: staff.EmployeeNumber,  
 Mobile: staff.Mobile,  
 PostalAddress: staff.PostalAddress,  
 Departments: staff.BusinessCategory,  
 Position: staff.DepartmentNumber,  
 Introduction: staff.CN,  
 Creator: "system",  
 Source: "openldap",  
 DepartmentId: tools.SliceToString(groupIds, ","),  
 SourceUserId: staff.Name,  
 SourceUnionId: staff.Name,  
 Roles: roles,  
 UserDN: staff.DN,  
 })  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("SyncOpenLdapUsers写入用户失败：%s", err.Error()))  
 }  
 }  
 return nil, nil  
}  
  
// AddUser 添加用户数据  
func (d OpenLdapLogic) AddUsers(user \*model.User) error {  
 // 根据 user\_dn 查询用户,不存在则创建  
 if !isql.User.Exist(tools.H{"user\_dn": user.UserDN}) {  
 if user.Departments == "" {  
 user.Departments = "默认:研发中心"  
 }  
 if user.GivenName == "" {  
 user.GivenName = user.Nickname  
 }  
 if user.PostalAddress == "" {  
 user.PostalAddress = "默认:地球"  
 }  
 if user.Position == "" {  
 user.Position = "默认:技术"  
 }  
 if user.Introduction == "" {  
 user.Introduction = user.Nickname  
 }  
 if user.JobNumber == "" {  
 user.JobNumber = "未启用"  
 }  
 // 先将用户添加到MySQL  
 err := isql.User.Add(user)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("向MySQL创建用户失败：" + err.Error()))  
 }  
  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 for \_, group := range groups {  
 if group.GroupDN[:3] == "ou=" {  
 continue  
 }  
 // 先将用户和部门信息维护到MySQL  
 err := isql.Group.AddUserToGroup(group, []model.User{\*user})  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("向MySQL添加用户到分组关系失败：" + err.Error()))  
 }  
 }  
 return nil  
 }  
 return nil  
}

package logic  
  
import (  
 "fmt"  
 "math/rand"  
 "time"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/common"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 jsoniter "github.com/json-iterator/go"  
 "github.com/robfig/cron/v3"  
 "github.com/tidwall/gjson"  
)  
  
var (  
 ReqAssertErr = tools.NewRspError(tools.SystemErr, fmt.Errorf("请求异常"))  
  
 Api = &ApiLogic{}  
 User = &UserLogic{}  
 Group = &GroupLogic{}  
 Role = &RoleLogic{}  
 Menu = &MenuLogic{}  
 OperationLog = &OperationLogLogic{}  
 DingTalk = &DingTalkLogic{}  
 WeCom = &WeComLogic{}  
 FeiShu = &FeiShuLogic{}  
 OpenLdap = &OpenLdapLogic{}  
 Sql = &SqlLogic{}  
 Base = &BaseLogic{}  
 FieldRelation = &FieldRelationLogic{}  
  
 json = jsoniter.ConfigCompatibleWithStandardLibrary  
)  
  
// CommonAddGroup 标准创建分组  
func CommonAddGroup(group \*model.Group) error {  
 // 先在ldap中创建组  
 err := ildap.Group.Add(group)  
 if err != nil {  
 return err  
 }  
  
 // 然后在数据库中创建组  
 err = isql.Group.Add(group)  
 if err != nil {  
 return err  
 }  
  
 // 默认创建分组之后，需要将admin添加到分组中  
 adminInfo := new(model.User)  
 err = isql.User.Find(tools.H{"id": 1}, adminInfo)  
 if err != nil {  
 return err  
 }  
  
 err = isql.Group.AddUserToGroup(group, []model.User{\*adminInfo})  
 if err != nil {  
 return err  
 }  
  
 return nil  
}  
  
// CommonUpdateGroup 标准更新分组  
func CommonUpdateGroup(oldGroup, newGroup \*model.Group) error {  
 //若配置了不允许修改分组名称，则不更新分组名称  
 if !config.Conf.Ldap.GroupNameModify {  
 newGroup.GroupName = oldGroup.GroupName  
 }  
  
 err := ildap.Group.Update(oldGroup, newGroup)  
 if err != nil {  
 return err  
 }  
 err = isql.Group.Update(newGroup)  
 if err != nil {  
 return err  
 }  
 return nil  
}  
  
// CommonAddUser 标准创建用户  
func CommonAddUser(user \*model.User, groups []\*model.Group) error {  
 // 用户信息的预置处理  
 if user.Nickname == "" {  
 user.Nickname = "佚名"  
 }  
 if user.GivenName == "" {  
 user.GivenName = user.Nickname  
 }  
 if user.Introduction == "" {  
 user.Introduction = user.Nickname  
 }  
 if user.Mail == "" {  
 // 兼容  
 if len(config.Conf.Ldap.DefaultEmailSuffix) > 0 {  
 user.Mail = user.Username + "@" + config.Conf.Ldap.DefaultEmailSuffix  
 } else {  
 user.Mail = user.Username + "@eryajf.net"  
 }  
 }  
 if user.JobNumber == "" {  
 user.JobNumber = "0000"  
 }  
 if user.Departments == "" {  
 user.Departments = "默认:研发中心"  
 }  
 if user.Position == "" {  
 user.Position = "默认:打工人"  
 }  
 if user.PostalAddress == "" {  
 user.PostalAddress = "默认:地球"  
 }  
 if user.Mobile == "" {  
 user.Mobile = generateMobile()  
 }  
  
 // 先将用户添加到MySQL  
 err := isql.User.Add(user)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("向MySQL创建用户失败：" + err.Error()))  
 }  
 // 再将用户添加到ldap  
 err = ildap.User.Add(user)  
 if err != nil {  
 return tools.NewLdapError(fmt.Errorf("AddUser向LDAP创建用户失败：" + err.Error()))  
 }  
  
 // 处理用户归属的组  
 for \_, group := range groups {  
 if group.GroupDN[:3] == "ou=" {  
 continue  
 }  
 // 先将用户和部门信息维护到MySQL  
 err := isql.Group.AddUserToGroup(group, []model.User{\*user})  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("向MySQL添加用户到分组关系失败：" + err.Error()))  
 }  
 //根据选择的部门，添加到部门内  
 err = ildap.Group.AddUserToGroup(group.GroupDN, user.UserDN)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("向Ldap添加用户到分组关系失败：" + err.Error()))  
 }  
 }  
 return nil  
}  
  
// CommonUpdateUser 标准更新用户  
func CommonUpdateUser(oldUser, newUser \*model.User, groupId []uint) error {  
 // 更新用户  
 if !config.Conf.Ldap.UserNameModify {  
 newUser.Username = oldUser.Username  
 }  
  
 err := ildap.User.Update(oldUser.Username, newUser)  
 if err != nil {  
 return tools.NewLdapError(fmt.Errorf("在LDAP更新用户失败：" + err.Error()))  
 }  
  
 err = isql.User.Update(newUser)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("在MySQL更新用户失败：" + err.Error()))  
 }  
  
 //判断部门信息是否有变化有变化则更新相应的数据库  
 oldDeptIds := tools.StringToSlice(oldUser.DepartmentId, ",")  
 addDeptIds, removeDeptIds := tools.ArrUintCmp(oldDeptIds, groupId)  
  
 // 先处理添加的部门  
 addgroups, err := isql.Group.GetGroupByIds(addDeptIds)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 for \_, group := range addgroups {  
 if group.GroupDN[:3] == "ou=" {  
 continue  
 }  
 // 先将用户和部门信息维护到MySQL  
 err := isql.Group.AddUserToGroup(group, []model.User{\*newUser})  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("向MySQL添加用户到分组关系失败：" + err.Error()))  
 }  
 //根据选择的部门，添加到部门内  
 err = ildap.Group.AddUserToGroup(group.GroupDN, newUser.UserDN)  
 if err != nil {  
 return tools.NewLdapError(fmt.Errorf("向Ldap添加用户到分组关系失败：" + err.Error()))  
 }  
 }  
  
 // 再处理删除的部门  
 removegroups, err := isql.Group.GetGroupByIds(removeDeptIds)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 for \_, group := range removegroups {  
 if group.GroupDN[:3] == "ou=" {  
 continue  
 }  
 err := isql.Group.RemoveUserFromGroup(group, []model.User{\*newUser})  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("在MySQL将用户从分组移除失败：" + err.Error()))  
 }  
 err = ildap.Group.RemoveUserFromGroup(group.GroupDN, newUser.UserDN)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("在ldap将用户从分组移除失败：" + err.Error()))  
 }  
 }  
 return nil  
}  
  
// BuildGroupData 根据数据与动态字段组装成分组数据  
func BuildGroupData(flag string, remoteData map[string]interface{}) (\*model.Group, error) {  
 output, err := json.Marshal(&remoteData)  
 if err != nil {  
 return nil, err  
 }  
  
 oldData := new(model.FieldRelation)  
 err = isql.FieldRelation.Find(tools.H{"flag": flag + "\_group"}, oldData)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
 frs, err := tools.JsonToMap(string(oldData.Attributes))  
 if err != nil {  
 return nil, tools.NewOperationError(err)  
 }  
  
 g := &model.Group{}  
 for system, remote := range frs {  
 switch system {  
 case "groupName":  
 g.SetGroupName(gjson.Get(string(output), remote).String())  
 case "remark":  
 g.SetRemark(gjson.Get(string(output), remote).String())  
 case "sourceDeptId":  
 g.SetSourceDeptId(fmt.Sprintf("%s\_%s", flag, gjson.Get(string(output), remote).String()))  
 case "sourceDeptParentId":  
 g.SetSourceDeptParentId(fmt.Sprintf("%s\_%s", flag, gjson.Get(string(output), remote).String()))  
 }  
 }  
 return g, nil  
}  
  
// BuildUserData 根据数据与动态字段组装成用户数据  
func BuildUserData(flag string, remoteData map[string]interface{}) (\*model.User, error) {  
 output, err := json.Marshal(&remoteData)  
 if err != nil {  
 return nil, err  
 }  
  
 fieldRelationSource := new(model.FieldRelation)  
 err = isql.FieldRelation.Find(tools.H{"flag": flag + "\_user"}, fieldRelationSource)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
 fieldRelation, err := tools.JsonToMap(string(fieldRelationSource.Attributes))  
 if err != nil {  
 return nil, tools.NewOperationError(err)  
 }  
  
 // 校验username是否为空，username为必填项  
 name := gjson.Get(string(output), fieldRelation["username"]).String()  
 if len(name) == 0 {  
 common.Log.Warnf("%s 该用户未填写username", output)  
 return nil, nil  
 }  
  
 u := &model.User{}  
 for system, remote := range fieldRelation {  
 switch system {  
 case "username":  
 u.SetUserName(gjson.Get(string(output), remote).String())  
 case "nickname":  
 u.SetNickName(gjson.Get(string(output), remote).String())  
 case "givenName":  
 u.SetGivenName(gjson.Get(string(output), remote).String())  
 case "mail":  
 u.SetMail(gjson.Get(string(output), remote).String())  
 case "jobNumber":  
 u.SetJobNumber(gjson.Get(string(output), remote).String())  
 case "mobile":  
 u.SetMobile(gjson.Get(string(output), remote).String())  
 case "avatar":  
 u.SetAvatar(gjson.Get(string(output), remote).String())  
 case "postalAddress":  
 u.SetPostalAddress(gjson.Get(string(output), remote).String())  
 case "position":  
 u.SetPosition(gjson.Get(string(output), remote).String())  
 case "introduction":  
 u.SetIntroduction(gjson.Get(string(output), remote).String())  
 case "sourceUserId":  
 u.SetSourceUserId(fmt.Sprintf("%s\_%s", flag, gjson.Get(string(output), remote).String()))  
 case "sourceUnionId":  
 u.SetSourceUnionId(fmt.Sprintf("%s\_%s", flag, gjson.Get(string(output), remote).String()))  
 }  
 }  
 return u, nil  
}  
  
// ConvertDeptData 将部门信息转成本地结构体  
func ConvertDeptData(flag string, remoteData []map[string]interface{}) (groups []\*model.Group, err error) {  
 for \_, dept := range remoteData {  
 group, err := BuildGroupData(flag, dept)  
 if err != nil {  
 return nil, err  
 }  
 groups = append(groups, group)  
 }  
 return  
}  
  
// ConvertUserData 将用户信息转成本地结构体  
func ConvertUserData(flag string, remoteData []map[string]interface{}) (users []\*model.User, err error) {  
 for \_, staff := range remoteData {  
 groupIds, err := isql.Group.DeptIdsToGroupIds(staff["department\_ids"].([]string))  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("将部门ids转换为内部部门id失败：%s", err.Error()))  
 }  
 user, err := BuildUserData(flag, staff)  
 if err != nil {  
 return nil, err  
 }  
 if user != nil {  
 user.DepartmentId = tools.SliceToString(groupIds, ",")  
 users = append(users, user)  
 }  
 }  
 return  
}  
  
func InitCron() {  
 c := cron.New(cron.WithSeconds())  
  
 if config.Conf.DingTalk.EnableSync {  
 //启动定时任务  
 \_, err := c.AddFunc(config.Conf.DingTalk.DeptSyncTime, func() {  
 DingTalk.SyncDingTalkDepts(nil, nil)  
 })  
 if err != nil {  
 common.Log.Errorf("启动同步部门的定时任务失败: %v", err)  
 }  
 //每天凌晨1点执行一次  
 \_, err = c.AddFunc(config.Conf.DingTalk.UserSyncTime, func() {  
 DingTalk.SyncDingTalkUsers(nil, nil)  
 })  
 if err != nil {  
 common.Log.Errorf("启动同步用户的定时任务失败: %v", err)  
 }  
 }  
 if config.Conf.WeCom.EnableSync {  
 \_, err := c.AddFunc(config.Conf.WeCom.DeptSyncTime, func() {  
 WeCom.SyncWeComDepts(nil, nil)  
 })  
 if err != nil {  
 common.Log.Errorf("启动同步部门的定时任务失败: %v", err)  
 }  
 //每天凌晨1点执行一次  
 \_, err = c.AddFunc(config.Conf.WeCom.UserSyncTime, func() {  
 WeCom.SyncWeComUsers(nil, nil)  
 })  
 if err != nil {  
 common.Log.Errorf("启动同步用户的定时任务失败: %v", err)  
 }  
 }  
 if config.Conf.FeiShu.EnableSync {  
 \_, err := c.AddFunc(config.Conf.FeiShu.DeptSyncTime, func() {  
 FeiShu.SyncFeiShuDepts(nil, nil)  
 })  
 if err != nil {  
 common.Log.Errorf("启动同步部门的定时任务失败: %v", err)  
 }  
 //每天凌晨1点执行一次  
 \_, err = c.AddFunc(config.Conf.FeiShu.UserSyncTime, func() {  
 FeiShu.SyncFeiShuUsers(nil, nil)  
 })  
 if err != nil {  
 common.Log.Errorf("启动同步用户的定时任务失败: %v", err)  
 }  
 }  
  
 // 自动检索未同步数据  
 \_, err := c.AddFunc("0 \*/2 \* \* \* \*", func() {  
 // 开发调试时调整为10秒执行一次  
 // \_, err := c.AddFunc("\*/10 \* \* \* \* \*", func() {  
 \_ = SearchGroupDiff()  
 \_ = SearchUserDiff()  
 })  
 if err != nil {  
 common.Log.Errorf("启动同步任务状态检查任务失败: %v", err)  
 }  
 c.Start()  
}  
  
func GroupListToTree(rootId string, groupList []\*model.Group) \*model.Group {  
 // 创建空根节点  
 rootGroup := &model.Group{SourceDeptId: rootId}  
 rootGroup.Children = groupListToTree(rootGroup, groupList)  
 return rootGroup  
}  
  
func groupListToTree(rootGroup \*model.Group, list []\*model.Group) []\*model.Group {  
 children := make([]\*model.Group, 0)  
 for \_, group := range list {  
 if group.SourceDeptParentId == rootGroup.SourceDeptId {  
 children = append(children, group)  
 }  
 }  
 for \_, group := range children {  
 group.Children = groupListToTree(group, list)  
 }  
 return children  
}  
  
func generateMobile() string {  
 rand.Seed(time.Now().UnixNano())  
 randNum := rand.Intn(9000000000) + 1000000000  
 randNum = randNum + 10000000000  
 if isql.User.Exist(tools.H{"mobile": randNum}) {  
 generateMobile()  
 }  
 return fmt.Sprintf("%v", randNum)  
}

package logic  
  
import (  
 "fmt"  
 "strings"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/client/dingtalk"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type DingTalkLogic struct {  
}  
  
// 通过钉钉获取部门信息  
func (d \*DingTalkLogic) SyncDingTalkDepts(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 // 1.获取所有部门  
 deptSource, err := dingtalk.GetAllDepts()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("获取钉钉部门列表失败：%s", err.Error()))  
 }  
 depts, err := ConvertDeptData(config.Conf.DingTalk.Flag, deptSource)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("转换钉钉部门数据失败：%s", err.Error()))  
 }  
  
 // 2.将远程数据转换成树  
 deptTree := GroupListToTree(fmt.Sprintf("%s\_1", config.Conf.DingTalk.Flag), depts)  
  
 // 3.根据树进行创建  
 err = d.addDepts(deptTree.Children)  
  
 return nil, err  
}  
  
// 添加部门  
func (d DingTalkLogic) addDepts(depts []\*model.Group) error {  
 for \_, dept := range depts {  
 err := d.AddDepts(dept)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("DsyncDingTalkDepts添加部门失败: %s", err.Error()))  
 }  
 if len(dept.Children) != 0 {  
 err = d.addDepts(dept.Children)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("DsyncDingTalkDepts添加部门失败: %s", err.Error()))  
 }  
 }  
 }  
 return nil  
}  
  
// AddGroup 添加部门数据  
func (d DingTalkLogic) AddDepts(group \*model.Group) error {  
 parentGroup := new(model.Group)  
 err := isql.Group.Find(tools.H{"source\_dept\_id": group.SourceDeptParentId}, parentGroup) // 查询当前分组父ID在MySQL中的数据信息  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("查询父级部门失败：%s", err.Error()))  
 }  
  
 // 此时的 group 已经附带了Build后动态关联好的字段，接下来将一些确定性的其他字段值添加上，就可以创建这个分组了  
 group.Creator = "system"  
 group.GroupType = "cn"  
 group.ParentId = parentGroup.ID  
 group.Source = config.Conf.DingTalk.Flag  
 group.GroupDN = fmt.Sprintf("cn=%s,%s", group.GroupName, parentGroup.GroupDN)  
  
 if !isql.Group.Exist(tools.H{"group\_dn": group.GroupDN}) { // 判断当前部门是否已落库  
 err = CommonAddGroup(group)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("添加部门: %s, 失败: %s", group.GroupName, err.Error()))  
 }  
 }  
 return nil  
}  
  
// 根据现有数据库同步到的部门信息，开启用户同步  
func (d DingTalkLogic) SyncDingTalkUsers(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 // 1.获取钉钉用户列表  
 staffSource, err := dingtalk.GetAllUsers()  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("SyncDingTalkUsers获取钉钉用户列表失败：%s", err.Error()))  
 }  
 staffs, err := ConvertUserData(config.Conf.DingTalk.Flag, staffSource)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("转换钉钉用户数据失败：%s", err.Error()))  
 }  
 // 2.遍历用户，开始写入  
 for \_, staff := range staffs {  
 // 入库  
 err = d.AddUsers(staff)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("SyncDingTalkUsers写入用户失败：%s", err.Error()))  
 }  
 }  
  
 // 3.获取钉钉已离职用户id列表  
 // 根据配置判断是查全部离职用户还是只查指定时间范围内的离职用户  
 var userIds []string  
 if config.Conf.DingTalk.ULeaveRange == 0 {  
 userIds, err = dingtalk.GetLeaveUserIds()  
 } else {  
 userIds, err = dingtalk.GetLeaveUserIdsDateRange(config.Conf.DingTalk.ULeaveRange)  
 }  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("SyncDingTalkUsers获取钉钉离职用户列表失败：%s", err.Error()))  
 }  
 // 4.遍历id，开始处理  
 for \_, uid := range userIds {  
 if isql.User.Exist(  
 tools.H{  
 "source\_user\_id": fmt.Sprintf("%s\_%s", config.Conf.DingTalk.Flag, uid),  
 "status": 1, //只处理1在职的  
 }) {  
 user := new(model.User)  
 err = isql.User.Find(tools.H{"source\_user\_id": fmt.Sprintf("%s\_%s", config.Conf.DingTalk.Flag, uid)}, user)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL查询用户失败: " + err.Error()))  
 }  
 // 先从ldap删除用户  
 err = ildap.User.Delete(user.UserDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("在LDAP删除用户失败" + err.Error()))  
 }  
 // 然后更新MySQL中用户状态  
 err = isql.User.ChangeStatus(int(user.ID), 2)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL更新用户状态失败: " + err.Error()))  
 }  
 }  
 }  
  
 return nil, nil  
}  
  
// AddUser 添加用户数据  
func (d DingTalkLogic) AddUsers(user \*model.User) error {  
 // 根据角色id获取角色  
 roles, err := isql.Role.GetRolesByIds([]uint{2}) // 默认添加为普通用户角色  
 if err != nil {  
 return tools.NewValidatorError(fmt.Errorf("根据角色ID获取角色信息失败:%s", err.Error()))  
 }  
 user.Roles = roles  
 user.Creator = "system"  
 user.Source = config.Conf.DingTalk.Flag  
 user.Password = config.Conf.Ldap.UserInitPassword  
 user.UserDN = fmt.Sprintf("uid=%s,%s", user.Username, config.Conf.Ldap.UserDN)  
  
 // 根据 user\_dn 查询用户,不存在则创建  
 if !isql.User.Exist(tools.H{"user\_dn": user.UserDN}) {  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 var deptTmp string  
 for \_, group := range groups {  
 deptTmp = deptTmp + group.GroupName + ","  
 }  
 user.Departments = strings.TrimRight(deptTmp, ",")  
  
 // 新增用户  
 err = CommonAddUser(user, groups)  
 if err != nil {  
 return tools.NewOperationError(fmt.Errorf("添加用户: %s, 失败: %s", user.Username, err.Error()))  
 }  
 } else {  
 if config.Conf.DingTalk.IsUpdateSyncd {  
 // 先获取用户信息  
 oldData := new(model.User)  
 err = isql.User.Find(tools.H{"user\_dn": user.UserDN}, oldData)  
 if err != nil {  
 return err  
 }  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
 var deptTmp string  
 for \_, group := range groups {  
 deptTmp = deptTmp + group.GroupName + ","  
 }  
 user.Model = oldData.Model  
 user.Roles = oldData.Roles  
 user.Creator = oldData.Creator  
 user.Source = oldData.Source  
 user.Password = oldData.Password  
 user.UserDN = oldData.UserDN  
 user.Departments = strings.TrimRight(deptTmp, ",")  
  
 // 用户信息的预置处理  
 if user.Nickname == "" {  
 user.Nickname = oldData.Nickname  
 }  
 if user.GivenName == "" {  
 user.GivenName = user.Nickname  
 }  
 if user.Introduction == "" {  
 user.Introduction = user.Nickname  
 }  
 if user.Mail == "" {  
 user.Mail = oldData.Mail  
 }  
 if user.JobNumber == "" {  
 user.JobNumber = oldData.JobNumber  
 }  
 if user.Departments == "" {  
 user.Departments = oldData.Departments  
 }  
 if user.Position == "" {  
 user.Position = oldData.Position  
 }  
 if user.PostalAddress == "" {  
 user.PostalAddress = oldData.PostalAddress  
 }  
 if user.Mobile == "" {  
 user.Mobile = oldData.Mobile  
 }  
 if err = CommonUpdateUser(oldData, user, tools.StringToSlice(user.DepartmentId, ",")); err != nil {  
 return err  
 }  
 }  
 }  
 return nil  
}

package logic  
  
import (  
 "fmt"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/model/response"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
)  
  
type BaseLogic struct{}  
  
// SendCode 发送验证码  
func (l BaseLogic) SendCode(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.BaseSendCodeReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 // 判断邮箱是否正确  
 user := new(model.User)  
 err := isql.User.Find(tools.H{"mail": r.Mail}, user)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("通过邮箱查询用户失败" + err.Error()))  
 }  
 if user.Status != 1 || user.SyncState != 1 {  
 return nil, tools.NewMySqlError(fmt.Errorf("该用户已离职或者未同步在ldap，无法重置密码，如有疑问，请联系管理员"))  
 }  
 err = tools.SendCode([]string{r.Mail})  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("邮件发送失败" + err.Error()))  
 }  
  
 return nil, nil  
}  
  
// ChangePwd 重置密码  
func (l BaseLogic) ChangePwd(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.BaseChangePwdReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 // 判断邮箱是否正确  
 if !isql.User.Exist(tools.H{"mail": r.Mail}) {  
 return nil, tools.NewValidatorError(fmt.Errorf("邮箱不存在,请检查邮箱是否正确"))  
 }  
 // 判断验证码是否过期  
 cacheCode, ok := tools.VerificationCodeCache.Get(r.Mail)  
 if !ok {  
 return nil, tools.NewValidatorError(fmt.Errorf("对不起，该验证码已超过5分钟有效期，请重新重新密码"))  
 }  
 // 判断验证码是否正确  
 if cacheCode != r.Code {  
 return nil, tools.NewValidatorError(fmt.Errorf("验证码错误，请检查邮箱中正确的验证码，如果点击多次发送验证码，请用最后一次生成的验证码来验证"))  
 }  
  
 user := new(model.User)  
 err := isql.User.Find(tools.H{"mail": r.Mail}, user)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("通过邮箱查询用户失败" + err.Error()))  
 }  
  
 newpass, err := ildap.User.NewPwd(user.Username)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("LDAP生成新密码失败" + err.Error()))  
 }  
  
 err = tools.SendMail([]string{user.Mail}, newpass)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("邮件发送失败" + err.Error()))  
 }  
  
 // 更新数据库密码  
 err = isql.User.ChangePwd(user.Username, tools.NewGenPasswd(newpass))  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL更新密码失败: " + err.Error()))  
 }  
  
 return nil, nil  
}  
  
// Dashboard 仪表盘  
func (l BaseLogic) Dashboard(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 \_, ok := req.(\*request.BaseDashboardReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 userCount, err := isql.User.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取用户总数失败"))  
 }  
 groupCount, err := isql.Group.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组总数失败"))  
 }  
 roleCount, err := isql.Role.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取角色总数失败"))  
 }  
 menuCount, err := isql.Menu.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取菜单总数失败"))  
 }  
 apiCount, err := isql.Api.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取接口总数失败"))  
 }  
 logCount, err := isql.OperationLog.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取日志总数失败"))  
 }  
  
 rst := make([]\*response.DashboardList, 0)  
  
 rst = append(rst,  
 &response.DashboardList{  
 DataType: "user",  
 DataName: "用户",  
 DataCount: userCount,  
 Icon: "people",  
 Path: "#/personnel/user",  
 },  
 &response.DashboardList{  
 DataType: "group",  
 DataName: "分组",  
 DataCount: groupCount,  
 Icon: "peoples",  
 Path: "#/personnel/group",  
 },  
 &response.DashboardList{  
 DataType: "role",  
 DataName: "角色",  
 DataCount: roleCount,  
 Icon: "eye-open",  
 Path: "#/system/role",  
 },  
 &response.DashboardList{  
 DataType: "menu",  
 DataName: "菜单",  
 DataCount: menuCount,  
 Icon: "tree-table",  
 Path: "#/system/menu",  
 },  
 &response.DashboardList{  
 DataType: "api",  
 DataName: "接口",  
 DataCount: apiCount,  
 Icon: "tree",  
 Path: "#/system/api",  
 },  
 &response.DashboardList{  
 DataType: "log",  
 DataName: "日志",  
 DataCount: logCount,  
 Icon: "documentation",  
 Path: "#/log/operation-log",  
 },  
 )  
  
 return rst, nil  
}  
  
// EncryptPasswd  
func (l BaseLogic) EncryptPasswd(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.EncryptPasswdReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 return tools.NewGenPasswd(r.Passwd), nil  
}  
  
// DecryptPasswd  
func (l BaseLogic) DecryptPasswd(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.DecryptPasswdReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 return tools.NewParPasswd(r.Passwd), nil  
}

package logic  
  
import (  
 "fmt"  
 "strconv"  
 "strings"  
  
 "github.com/eryajf/go-ldap-admin/config"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/model/response"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
)  
  
type GroupLogic struct{}  
  
// Add 添加数据  
func (l GroupLogic) Add(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.GroupAddReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 // 获取当前用户  
 ctxUser, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前登陆用户信息失败"))  
 }  
  
 group := model.Group{  
 GroupType: r.GroupType,  
 ParentId: r.ParentId,  
 GroupName: r.GroupName,  
 Remark: r.Remark,  
 Creator: ctxUser.Username,  
 Source: "platform", //默认是平台添加  
 }  
  
 if r.ParentId == 0 {  
 group.SourceDeptId = "platform\_0"  
 group.SourceDeptParentId = "platform\_0"  
 group.GroupDN = fmt.Sprintf("%s=%s,%s", r.GroupType, r.GroupName, config.Conf.Ldap.BaseDN)  
 } else {  
 parentGroup := new(model.Group)  
 err := isql.Group.Find(tools.H{"id": r.ParentId}, parentGroup)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取父级组信息失败"))  
 }  
 group.SourceDeptId = "platform\_0"  
 group.SourceDeptParentId = fmt.Sprintf("%s\_%d", parentGroup.Source, r.ParentId)  
 group.GroupDN = fmt.Sprintf("%s=%s,%s", r.GroupType, r.GroupName, parentGroup.GroupDN)  
 }  
  
 // 根据 group\_dn 判断分组是否已存在  
 if isql.Group.Exist(tools.H{"group\_dn": group.GroupDN}) {  
 return nil, tools.NewValidatorError(fmt.Errorf("该分组对应DN已存在"))  
 }  
  
 // 先在ldap中创建组  
 err = ildap.Group.Add(&group)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("向LDAP创建分组失败" + err.Error()))  
 }  
  
 // 然后在数据库中创建组  
 err = isql.Group.Add(&group)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("向MySQL创建分组失败"))  
 }  
  
 // 默认创建分组之后，需要将admin添加到分组中  
 adminInfo := new(model.User)  
 err = isql.User.Find(tools.H{"id": 1}, adminInfo)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
  
 err = isql.Group.AddUserToGroup(&group, []model.User{\*adminInfo})  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("添加用户到分组失败: %s", err.Error()))  
 }  
  
 return nil, nil  
}  
  
// List 数据列表  
func (l GroupLogic) List(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.GroupListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 // 获取数据列表  
 groups, err := isql.Group.List(r)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组列表失败: %s", err.Error()))  
 }  
  
 rets := make([]model.Group, 0)  
 for \_, group := range groups {  
 rets = append(rets, \*group)  
 }  
 count, err := isql.Group.Count()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组总数失败"))  
 }  
  
 return response.GroupListRsp{  
 Total: count,  
 Groups: rets,  
 }, nil  
}  
  
// GetTree 数据树  
func (l GroupLogic) GetTree(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.GroupListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 var groups []\*model.Group  
 groups, err := isql.Group.ListTree(r)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取资源列表失败: " + err.Error()))  
 }  
  
 tree := isql.GenGroupTree(0, groups)  
  
 return tree, nil  
}  
  
// Update 更新数据  
func (l GroupLogic) Update(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.GroupUpdateReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"id": int(r.ID)}  
 if !isql.Group.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("分组不存在"))  
 }  
  
 // 获取当前登陆用户  
 ctxUser, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前登陆用户失败"))  
 }  
  
 oldGroup := new(model.Group)  
 err = isql.Group.Find(filter, oldGroup)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
  
 newGroup := model.Group{  
 Model: oldGroup.Model,  
 GroupName: r.GroupName,  
 Remark: r.Remark,  
 Creator: ctxUser.Username,  
 GroupType: oldGroup.GroupType,  
 }  
  
 //若配置了不允许修改分组名称，则不更新分组名称  
 if !config.Conf.Ldap.GroupNameModify {  
 newGroup.GroupName = oldGroup.GroupName  
 }  
  
 err = ildap.Group.Update(oldGroup, &newGroup)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("向LDAP更新分组失败：" + err.Error()))  
 }  
 err = isql.Group.Update(&newGroup)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("向MySQL更新分组失败"))  
 }  
 return nil, nil  
}  
  
// Delete 删除数据  
func (l GroupLogic) Delete(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.GroupDeleteReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 for \_, id := range r.GroupIds {  
 filter := tools.H{"id": int(id)}  
 if !isql.Group.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("有分组不存在"))  
 }  
 }  
  
 groups, err := isql.Group.GetGroupByIds(r.GroupIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组列表失败: %s", err.Error()))  
 }  
  
 for \_, group := range groups {  
 // 判断存在子分组，不允许删除  
 filter := tools.H{"parent\_id": int(group.ID)}  
 if isql.Group.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("存在子分组，请先删除子分组，再执行该分组的删除操作！"))  
 }  
  
 // 删除的时候先从ldap进行删除  
 err = ildap.Group.Delete(group.GroupDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("向LDAP删除分组失败：" + err.Error()))  
 }  
 }  
  
 // 从MySQL中删除  
 err = isql.Group.Delete(groups)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("删除接口失败: %s", err.Error()))  
 }  
  
 return nil, nil  
}  
  
// AddUser 添加用户到分组  
func (l GroupLogic) AddUser(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.GroupAddUserReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"id": r.GroupID}  
  
 if !isql.Group.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("分组不存在"))  
 }  
  
 users, err := isql.User.GetUserByIds(r.UserIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取用户列表失败: %s", err.Error()))  
 }  
  
 group := new(model.Group)  
 err = isql.Group.Find(filter, group)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组失败: %s", err.Error()))  
 }  
  
 if group.GroupDN[:3] == "ou=" {  
 return nil, tools.NewMySqlError(fmt.Errorf("ou类型的分组不能添加用户"))  
 }  
  
 // 先添加到MySQL  
 err = isql.Group.AddUserToGroup(group, users)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("添加用户到分组失败: %s", err.Error()))  
 }  
  
 // 再往ldap添加  
 for \_, user := range users {  
 err = ildap.Group.AddUserToGroup(group.GroupDN, user.UserDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("向LDAP添加用户到分组失败" + err.Error()))  
 }  
 }  
  
 for \_, user := range users {  
 oldData := new(model.User)  
 err = isql.User.Find(tools.H{"id": user.ID}, oldData)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
 newData := oldData  
 // 添加新增的分组ID与部门  
 newData.DepartmentId = oldData.DepartmentId + "," + strconv.Itoa(int(r.GroupID))  
 newData.Departments = oldData.Departments + "," + group.GroupName  
 err = l.updataUser(newData)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("处理用户的部门数据失败:" + err.Error()))  
 }  
 }  
  
 return nil, nil  
}  
  
func (l GroupLogic) updataUser(newUser \*model.User) error {  
 err := isql.User.Update(newUser)  
 if err != nil {  
 return tools.NewMySqlError(fmt.Errorf("在MySQL更新用户失败：" + err.Error()))  
 }  
 return nil  
}  
  
// RemoveUser 移除用户  
func (l GroupLogic) RemoveUser(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.GroupRemoveUserReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"id": r.GroupID}  
  
 if !isql.Group.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("分组不存在"))  
 }  
  
 users, err := isql.User.GetUserByIds(r.UserIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取用户列表失败: %s", err.Error()))  
 }  
  
 group := new(model.Group)  
 err = isql.Group.Find(filter, group)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组失败: %s", err.Error()))  
 }  
  
 if group.GroupDN[:3] == "ou=" {  
 return nil, tools.NewMySqlError(fmt.Errorf("ou类型的分组内没有用户"))  
 }  
  
 // 先操作ldap  
 for \_, user := range users {  
 err := ildap.Group.RemoveUserFromGroup(group.GroupDN, user.UserDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("将用户从ldap移除失败" + err.Error()))  
 }  
 }  
  
 // 再操作MySQL  
 err = isql.Group.RemoveUserFromGroup(group, users)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("将用户从MySQL移除失败: %s", err.Error()))  
 }  
  
 for \_, user := range users {  
 oldData := new(model.User)  
 err = isql.User.Find(tools.H{"id": user.ID}, oldData)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
 newData := oldData  
  
 var newDepts []string  
 var newDeptIds []string  
 // 删掉移除的分组名字  
 for \_, v := range strings.Split(oldData.Departments, ",") {  
 if v != group.GroupName {  
 newDepts = append(newDepts, v)  
 }  
 }  
 // 删掉移除的分组id  
 for \_, v := range strings.Split(oldData.DepartmentId, ",") {  
 if v != strconv.Itoa(int(r.GroupID)) {  
 newDeptIds = append(newDeptIds, v)  
 }  
 }  
  
 newData.Departments = strings.Join(newDepts, ",")  
 newData.DepartmentId = strings.Join(newDeptIds, ",")  
 err = l.updataUser(newData)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("处理用户的部门数据失败:" + err.Error()))  
 }  
 }  
  
 return nil, nil  
}  
  
// UserInGroup 在分组内的用户  
func (l GroupLogic) UserInGroup(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserInGroupReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"id": r.GroupID}  
  
 if !isql.Group.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("分组不存在"))  
 }  
  
 group := new(model.Group)  
 err := isql.Group.Find(filter, group)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组失败: %s", err.Error()))  
 }  
  
 rets := make([]response.Guser, 0)  
  
 for \_, user := range group.Users {  
 if r.Nickname != "" && !strings.Contains(user.Nickname, r.Nickname) {  
 continue  
 }  
 rets = append(rets, response.Guser{  
 UserId: int64(user.ID),  
 UserName: user.Username,  
 NickName: user.Nickname,  
 Mail: user.Mail,  
 JobNumber: user.JobNumber,  
 Mobile: user.Mobile,  
 Introduction: user.Introduction,  
 })  
 }  
  
 return response.GroupUsers{  
 GroupId: int64(group.ID),  
 GroupName: group.GroupName,  
 GroupRemark: group.Remark,  
 UserList: rets,  
 }, nil  
}  
  
// UserNoInGroup 不在分组内的用户  
func (l GroupLogic) UserNoInGroup(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserNoInGroupReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 filter := tools.H{"id": r.GroupID}  
  
 if !isql.Group.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("分组不存在"))  
 }  
  
 group := new(model.Group)  
 err := isql.Group.Find(filter, group)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取分组失败: %s", err.Error()))  
 }  
  
 var userList []\*model.User  
 userList, err = isql.User.ListAll()  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取资源列表失败: " + err.Error()))  
 }  
  
 rets := make([]response.Guser, 0)  
 for \_, user := range userList {  
 in := true  
 for \_, groupUser := range group.Users {  
 if user.Username == groupUser.Username {  
 in = false  
 break  
 }  
 }  
 if in {  
 if r.Nickname != "" && !strings.Contains(user.Nickname, r.Nickname) {  
 continue  
 }  
 rets = append(rets, response.Guser{  
 UserId: int64(user.ID),  
 UserName: user.Username,  
 NickName: user.Nickname,  
 Mail: user.Mail,  
 JobNumber: user.JobNumber,  
 Mobile: user.Mobile,  
 Introduction: user.Introduction,  
 })  
 }  
 }  
  
 return response.GroupUsers{  
 GroupId: int64(group.ID),  
 GroupName: group.GroupName,  
 GroupRemark: group.Remark,  
 UserList: rets,  
 }, nil  
}

package logic  
  
import (  
 "fmt"  
 "strings"  
  
 "github.com/eryajf/go-ldap-admin/config"  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/eryajf/go-ldap-admin/model/response"  
 "github.com/eryajf/go-ldap-admin/public/tools"  
 "github.com/eryajf/go-ldap-admin/service/ildap"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
  
 "github.com/gin-gonic/gin"  
 "github.com/thoas/go-funk"  
)  
  
type UserLogic struct{}  
  
// Add 添加数据  
func (l UserLogic) Add(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserAddReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 if isql.User.Exist(tools.H{"username": r.Username}) {  
 return nil, tools.NewValidatorError(fmt.Errorf("用户名已存在,请勿重复添加"))  
 }  
 if isql.User.Exist(tools.H{"mobile": r.Mobile}) {  
 return nil, tools.NewValidatorError(fmt.Errorf("手机号已存在,请勿重复添加"))  
 }  
 if isql.User.Exist(tools.H{"job\_number": r.JobNumber}) {  
 return nil, tools.NewValidatorError(fmt.Errorf("工号已存在,请勿重复添加"))  
 }  
 if isql.User.Exist(tools.H{"mail": r.Mail}) {  
 return nil, tools.NewValidatorError(fmt.Errorf("邮箱已存在,请勿重复添加"))  
 }  
  
 // 密码通过RSA解密  
 // 密码不为空就解密  
 if r.Password != "" {  
 decodeData, err := tools.RSADecrypt([]byte(r.Password), config.Conf.System.RSAPrivateBytes)  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("密码解密失败"))  
 }  
 r.Password = string(decodeData)  
 if len(r.Password) < 6 {  
 return nil, tools.NewValidatorError(fmt.Errorf("密码长度至少为6位"))  
 }  
 } else {  
 r.Password = config.Conf.Ldap.UserInitPassword  
 }  
  
 // 当前登陆用户角色排序最小值（最高等级角色）以及当前登陆的用户  
 currentRoleSortMin, ctxUser, err := isql.User.GetCurrentUserMinRoleSort(c)  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("获取当前登陆用户角色排序最小值失败"))  
 }  
  
 // 根据角色id获取角色  
 if r.RoleIds == nil || len(r.RoleIds) == 0 {  
 r.RoleIds = []uint{2} // 默认添加为普通用户角色  
 }  
  
 roles, err := isql.Role.GetRolesByIds(r.RoleIds)  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("根据角色ID获取角色信息失败"))  
 }  
  
 var reqRoleSorts []int  
 for \_, role := range roles {  
 reqRoleSorts = append(reqRoleSorts, int(role.Sort))  
 }  
 // 前端传来用户角色排序最小值（最高等级角色）  
 reqRoleSortMin := uint(funk.MinInt(reqRoleSorts).(int))  
  
 // 如果登录用户的角色ID为1，亦即为管理员，则直接放行，保障管理员拥有最大权限  
 if currentRoleSortMin != 1 {  
 // 当前用户的角色排序最小值 需要小于 前端传来的角色排序最小值（用户不能创建比自己等级高的或者相同等级的用户）  
 if currentRoleSortMin >= reqRoleSortMin {  
 return nil, tools.NewValidatorError(fmt.Errorf("用户不能创建比自己等级高的或者相同等级的用户"))  
 }  
 }  
 user := model.User{  
 Username: r.Username,  
 Password: r.Password,  
 Nickname: r.Nickname,  
 GivenName: r.GivenName,  
 Mail: r.Mail,  
 JobNumber: r.JobNumber,  
 Mobile: r.Mobile,  
 Avatar: r.Avatar,  
 PostalAddress: r.PostalAddress,  
 Departments: r.Departments,  
 Position: r.Position,  
 Introduction: r.Introduction,  
 Status: r.Status,  
 Creator: ctxUser.Username,  
 DepartmentId: tools.SliceToString(r.DepartmentId, ","),  
 Source: r.Source,  
 Roles: roles,  
 UserDN: fmt.Sprintf("uid=%s,%s", r.Username, config.Conf.Ldap.UserDN),  
 }  
  
 if user.Source == "" {  
 user.Source = "platform"  
 }  
  
 // 获取用户将要添加的分组  
 groups, err := isql.Group.GetGroupByIds(tools.StringToSlice(user.DepartmentId, ","))  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("根据部门ID获取部门信息失败" + err.Error()))  
 }  
  
 err = CommonAddUser(&user, groups)  
 if err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("添加用户失败" + err.Error()))  
 }  
 return nil, nil  
}  
  
// List 数据列表  
func (l UserLogic) List(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserListReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 users, err := isql.User.List(r)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取用户列表失败：" + err.Error()))  
 }  
  
 rets := make([]model.User, 0)  
 for \_, user := range users {  
 rets = append(rets, \*user)  
 }  
 count, err := isql.User.ListCount(r)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取用户总数失败：" + err.Error()))  
 }  
  
 return response.UserListRsp{  
 Total: int(count),  
 Users: rets,  
 }, nil  
}  
  
// Update 更新数据  
func (l UserLogic) Update(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserUpdateReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 if !isql.User.Exist(tools.H{"id": r.ID}) {  
 return nil, tools.NewMySqlError(fmt.Errorf("该记录不存在"))  
 }  
  
 // 获取当前登陆用户  
 ctxUser, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前登陆用户失败"))  
 }  
  
 // 获取当前登陆用户角色ID集合  
 var currentRoleSorts []int  
 for \_, role := range ctxUser.Roles {  
 currentRoleSorts = append(currentRoleSorts, int(role.Sort))  
 }  
  
 // 获取将要操作的用户角色ID集合  
 var reqRoleSorts []int  
 roles, \_ := isql.Role.GetRolesByIds(r.RoleIds)  
 if len(roles) == 0 {  
 return nil, tools.NewValidatorError(fmt.Errorf("根据角色ID获取角色信息失败"))  
 }  
 for \_, role := range roles {  
 reqRoleSorts = append(reqRoleSorts, int(role.Sort))  
 }  
  
 // 当前登陆用户角色排序最小值（最高等级角色）  
 currentRoleSortMin := funk.MinInt(currentRoleSorts).(int)  
 // 前端传来用户角色排序最小值（最高等级角色）  
 reqRoleSortMin := funk.MinInt(reqRoleSorts).(int)  
  
 // 如果登录用户的角色ID为1，亦即为管理员，则直接放行，保障管理员拥有最大权限  
 if currentRoleSortMin != 1 {  
 // 判断是更新自己还是更新别人,如果操作的ID与登陆用户的ID一致，则说明操作的是自己  
 if int(r.ID) == int(ctxUser.ID) {  
 // 不能更改自己的角色  
 reqDiff, currentDiff := funk.Difference(reqRoleSorts, currentRoleSorts)  
 if len(reqDiff.([]int)) > 0 || len(currentDiff.([]int)) > 0 {  
 return nil, tools.NewValidatorError(fmt.Errorf("不能更改自己的角色"))  
 }  
 }  
  
 // 如果是更新别人，操作者不能更新比自己角色等级高的或者相同等级的用户  
 minRoleSorts, err := isql.User.GetUserMinRoleSortsByIds([]uint{uint(r.ID)}) // 根据userIdID获取用户角色排序最小值  
 if err != nil || len(minRoleSorts) == 0 {  
 return nil, tools.NewValidatorError(fmt.Errorf("根据用户ID获取用户角色排序最小值失败"))  
 }  
 if currentRoleSortMin >= minRoleSorts[0] || currentRoleSortMin >= reqRoleSortMin {  
 return nil, tools.NewValidatorError(fmt.Errorf("用户不能更新比自己角色等级高的或者相同等级的用户"))  
 }  
 }  
  
 // 先获取用户信息  
 oldData := new(model.User)  
 err = isql.User.Find(tools.H{"id": r.ID}, oldData)  
 if err != nil {  
 return nil, tools.NewMySqlError(err)  
 }  
  
 // 过滤掉前端会选择到的 请选择部门信息 这个选项  
 var (  
 depts string  
 deptids []uint  
 )  
 for \_, v := range strings.Split(r.Departments, ",") {  
 if v != "请选择部门信息" {  
 depts += v + ","  
 }  
 }  
 for \_, j := range r.DepartmentId {  
 if j != 0 {  
 deptids = append(deptids, j)  
 }  
 }  
  
 // 拼装新的用户信息  
 user := model.User{  
 Model: oldData.Model,  
 Username: r.Username,  
 Nickname: r.Nickname,  
 GivenName: r.GivenName,  
 Mail: r.Mail,  
 JobNumber: r.JobNumber,  
 Mobile: r.Mobile,  
 Avatar: r.Avatar,  
 PostalAddress: r.PostalAddress,  
 Departments: depts,  
 Position: r.Position,  
 Introduction: r.Introduction,  
 Creator: ctxUser.Username,  
 DepartmentId: tools.SliceToString(deptids, ","),  
 Source: oldData.Source,  
 Roles: roles,  
 UserDN: oldData.UserDN,  
 }  
  
 if err = CommonUpdateUser(oldData, &user, r.DepartmentId); err != nil {  
 return nil, tools.NewOperationError(fmt.Errorf("更新用户失败" + err.Error()))  
 }  
  
 return nil, nil  
}  
  
// Delete 删除数据  
func (l UserLogic) Delete(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserDeleteReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
  
 for \_, id := range r.UserIds {  
 filter := tools.H{"id": int(id)}  
 if !isql.User.Exist(filter) {  
 return nil, tools.NewMySqlError(fmt.Errorf("有用户不存在"))  
 }  
 }  
  
 // 根据用户ID获取用户角色排序最小值  
 roleMinSortList, err := isql.User.GetUserMinRoleSortsByIds(r.UserIds)  
 if err != nil || len(roleMinSortList) == 0 {  
 return nil, tools.NewValidatorError(fmt.Errorf("根据用户ID获取用户角色排序最小值失败"))  
 }  
  
 // 获取当前登陆用户角色排序最小值（最高等级角色）以及当前用户  
 minSort, ctxUser, err := isql.User.GetCurrentUserMinRoleSort(c)  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("获取当前登陆用户角色排序最小值失败"))  
 }  
  
 // 不能删除自己  
 if funk.Contains(r.UserIds, ctxUser.ID) {  
 return nil, tools.NewValidatorError(fmt.Errorf("用户不能删除自己"))  
 }  
  
 // 不能删除比自己(登陆用户)角色排序低(等级高)的用户  
 for \_, sort := range roleMinSortList {  
 if int(minSort) > sort {  
 return nil, tools.NewValidatorError(fmt.Errorf("用户不能删除比自己角色等级高的用户"))  
 }  
 }  
  
 users, err := isql.User.GetUserByIds(r.UserIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取用户信息失败: " + err.Error()))  
 }  
  
 // 先将用户从ldap中删除  
 for \_, user := range users {  
 err := ildap.User.Delete(user.UserDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("在LDAP删除用户失败" + err.Error()))  
 }  
 }  
  
 // 再将用户从MySQL中删除  
 err = isql.User.Delete(r.UserIds)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL删除用户失败: " + err.Error()))  
 }  
  
 return nil, nil  
}  
  
// ChangePwd 修改密码  
func (l UserLogic) ChangePwd(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserChangePwdReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 // 前端传来的密码是rsa加密的,先解密  
 // 密码通过RSA解密  
 decodeOldPassword, err := tools.RSADecrypt([]byte(r.OldPassword), config.Conf.System.RSAPrivateBytes)  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("原密码解析失败"))  
 }  
 decodeNewPassword, err := tools.RSADecrypt([]byte(r.NewPassword), config.Conf.System.RSAPrivateBytes)  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("新密码解析失败"))  
 }  
 r.OldPassword = string(decodeOldPassword)  
 r.NewPassword = string(decodeNewPassword)  
 // 获取当前用户  
 user, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前登陆用户失败"))  
 }  
 // 获取用户的真实正确密码  
 // correctPasswd := user.Password  
 // 判断前端请求的密码是否等于真实密码  
 // err = tools.ComparePasswd(correctPasswd, r.OldPassword)  
 // if err != nil {  
 // return nil, tools.NewValidatorError(fmt.Errorf("原密码错误"))  
 // }  
 if tools.NewParPasswd(user.Password) != r.OldPassword {  
 return nil, tools.NewValidatorError(fmt.Errorf("原密码错误"))  
 }  
 // ldap更新密码时可以直接指定用户DN和新密码即可更改成功  
 err = ildap.User.ChangePwd(user.UserDN, "", r.NewPassword)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("在LDAP更新密码失败" + err.Error()))  
 }  
  
 // 更新密码  
 err = isql.User.ChangePwd(user.Username, tools.NewGenPasswd(r.NewPassword))  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL更新密码失败: " + err.Error()))  
 }  
  
 return nil, nil  
}  
  
// ChangeUserStatus 修改用户状态  
func (l UserLogic) ChangeUserStatus(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserChangeUserStatusReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
 \_ = c  
 // 校验工作  
 filter := tools.H{"id": r.ID}  
 if !isql.User.Exist(filter) {  
 return nil, tools.NewValidatorError(fmt.Errorf("该用户不存在"))  
 }  
 user := new(model.User)  
 err := isql.User.Find(filter, user)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL查询用户失败: " + err.Error()))  
 }  
  
 if r.Status == 1 && r.Status == user.Status {  
 return nil, tools.NewValidatorError(fmt.Errorf("用户已经是在职状态"))  
 }  
 if r.Status == 2 && r.Status == user.Status {  
 return nil, tools.NewValidatorError(fmt.Errorf("用户已经是离职状态"))  
 }  
  
 // 获取当前登录用户，只有管理员才能够将用户状态改变  
 // 获取当前登陆用户角色排序最小值（最高等级角色）以及当前用户  
 minSort, \_, err := isql.User.GetCurrentUserMinRoleSort(c)  
 if err != nil {  
 return nil, tools.NewValidatorError(fmt.Errorf("获取当前登陆用户角色排序最小值失败"))  
 }  
  
 if int(minSort) != 1 {  
 return nil, tools.NewValidatorError(fmt.Errorf("只有管理员才能更改用户状态"))  
 }  
  
 if r.Status == 2 {  
 err = ildap.User.Delete(user.UserDN)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("在LDAP删除用户失败" + err.Error()))  
 }  
 } else {  
 err = ildap.User.Add(user)  
 if err != nil {  
 return nil, tools.NewLdapError(fmt.Errorf("在LDAP添加用户失败" + err.Error()))  
 }  
 }  
 err = isql.User.ChangeStatus(int(r.ID), int(r.Status))  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("在MySQL更新用户状态失败: " + err.Error()))  
 }  
 return nil, nil  
}  
  
// GetUserInfo 获取用户信息  
func (l UserLogic) GetUserInfo(c \*gin.Context, req interface{}) (data interface{}, rspError interface{}) {  
 r, ok := req.(\*request.UserGetUserInfoReq)  
 if !ok {  
 return nil, ReqAssertErr  
 }  
  
 \_ = c  
 \_ = r  
  
 user, err := isql.User.GetCurrentLoginUser(c)  
 if err != nil {  
 return nil, tools.NewMySqlError(fmt.Errorf("获取当前用户信息失败: " + err.Error()))  
 }  
 return user, nil  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/gin-gonic/gin"  
)  
  
type PasswordPolicyController struct{}  
  
var PasswordPolicy = new(PasswordPolicyController)  
  
// UpdatePolicy 更新密码策略  
// @Summary 更新密码策略  
// @Description 更新系统密码策略配置  
// @Tags 密码策略管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.PasswordPolicyReq true "密码策略信息"  
// @Success 200 {object} response.ResponseBody  
// @Router /password-policy/update [post]  
func (m \*PasswordPolicyController) UpdatePolicy(c \*gin.Context) {  
 req := new(request.PasswordPolicyReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.PasswordPolicy.UpdatePolicy(c, req)  
 })  
}  
  
// GetPolicy 获取密码策略  
// @Summary 获取密码策略  
// @Description 获取当前系统密码策略配置  
// @Tags 密码策略管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /password-policy/get [get]  
func (m \*PasswordPolicyController) GetPolicy(c \*gin.Context) {  
 Run(c, nil, func() (interface{}, interface{}) {  
 return logic.PasswordPolicy.GetPolicy(c)  
 })  
}

package controller  
  
import (  
 "fmt"  
 "net/http"  
 "regexp"  
  
 "github.com/eryajf/go-ldap-admin/public/tools"  
  
 "github.com/gin-gonic/gin"  
 "github.com/go-playground/locales/zh"  
 ut "github.com/go-playground/universal-translator"  
 "github.com/go-playground/validator/v10"  
 zht "github.com/go-playground/validator/v10/translations/zh"  
)  
  
var (  
 Api = &ApiController{}  
 Group = &GroupController{}  
 Menu = &MenuController{}  
 Role = &RoleController{}  
 User = &UserController{}  
 OperationLog = &OperationLogController{}  
 Base = &BaseController{}  
 FieldRelation = &FieldRelationController{}  
  
 validate = validator.New()  
 trans ut.Translator  
)  
  
func init() {  
 uni := ut.New(zh.New())  
 trans, \_ = uni.GetTranslator("zh")  
 \_ = zht.RegisterDefaultTranslations(validate, trans)  
 \_ = validate.RegisterValidation("checkMobile", checkMobile)  
}  
  
func checkMobile(fl validator.FieldLevel) bool {  
 reg := `1\d{10}`  
 rgx := regexp.MustCompile(reg)  
 return rgx.MatchString(fl.Field().String())  
}  
  
func Run(c \*gin.Context, req interface{}, fn func() (interface{}, interface{})) {  
 var err error  
 // bind struct  
 err = c.Bind(req)  
 if err != nil {  
 tools.Err(c, tools.NewValidatorError(err), nil)  
 return  
 }  
 // 校验  
 err = validate.Struct(req)  
 if err != nil {  
 for \_, err := range err.(validator.ValidationErrors) {  
 tools.Err(c, tools.NewValidatorError(fmt.Errorf(err.Translate(trans))), nil)  
 return  
 }  
 }  
 data, err1 := fn()  
 if err1 != nil {  
 tools.Err(c, tools.ReloadErr(err1), data)  
 return  
 }  
 tools.Success(c, data)  
}  
  
// Demo  
// @Summary 健康检测  
// @Tags 基础管理  
// @Produce json  
// @Description 健康检测  
// @Success 200 {object} response.ResponseBody  
// @router /base/ping [get]  
func Demo(c \*gin.Context) {  
 // 健康检测  
 CodeDebug()  
 c.JSON(http.StatusOK, tools.H{"code": 200, "msg": "ok", "data": "pong"})  
}  
  
func CodeDebug() {  
}

package controller  
  
import (  
 "strconv"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/response"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type ichActivityController struct{}  
  
var ICHActivity = new(ichActivityController)  
  
// Add 添加活动记录  
func (m \*ichActivityController) Add(c \*gin.Context) {  
 var activity model.ICHActivity  
 err := c.ShouldBindJSON(&activity)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 err = isql.ICHActivity.Add(&activity)  
 if err != nil {  
 response.Failed(c, "添加失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, "添加成功")  
}  
  
// Update 更新活动记录  
func (m \*ichActivityController) Update(c \*gin.Context) {  
 var activity model.ICHActivity  
 err := c.ShouldBindJSON(&activity)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 err = isql.ICHActivity.Update(&activity)  
 if err != nil {  
 response.Failed(c, "更新失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, "更新成功")  
}  
  
// Delete 删除活动记录  
func (m \*ichActivityController) Delete(c \*gin.Context) {  
 id := c.Param("id")  
 idUint, err := strconv.ParseUint(id, 10, 32)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 err = isql.ICHActivity.Delete(uint(idUint))  
 if err != nil {  
 response.Failed(c, "删除失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, "删除成功")  
}  
  
// GetByID 根据ID获取活动记录  
func (m \*ichActivityController) GetByID(c \*gin.Context) {  
 id := c.Param("id")  
 idUint, err := strconv.ParseUint(id, 10, 32)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 activity, err := isql.ICHActivity.GetByID(uint(idUint))  
 if err != nil {  
 response.Failed(c, "获取失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, activity)  
}  
  
// List 获取活动记录列表  
func (m \*ichActivityController) List(c \*gin.Context) {  
 page, \_ := strconv.Atoi(c.DefaultQuery("page", "1"))  
 size, \_ := strconv.Atoi(c.DefaultQuery("size", "10"))  
 query := c.DefaultQuery("query", "")  
  
 activities, total, err := isql.ICHActivity.List(page, size, query)  
 if err != nil {  
 response.Failed(c, "获取失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, gin.H{  
 "items": activities,  
 "total": total,  
 })  
}  
  
// GetByInheritorID 获取传承人的活动记录  
func (m \*ichActivityController) GetByInheritorID(c \*gin.Context) {  
 inheritorID := c.Query("inheritorId")  
 if inheritorID == "" {  
 response.Failed(c, "传承人ID不能为空")  
 return  
 }  
  
 idUint, err := strconv.ParseUint(inheritorID, 10, 32)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 activities, err := isql.ICHActivity.GetByInheritorID(uint(idUint))  
 if err != nil {  
 response.Failed(c, "获取失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, activities)  
}  
  
// GetStatistics 获取活动统计信息  
func (m \*ichActivityController) GetStatistics(c \*gin.Context) {  
 inheritorID := c.Query("inheritorId")  
 if inheritorID == "" {  
 response.Failed(c, "传承人ID不能为空")  
 return  
 }  
  
 idUint, err := strconv.ParseUint(inheritorID, 10, 32)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 stats, err := isql.ICHActivity.GetStatistics(uint(idUint))  
 if err != nil {  
 response.Failed(c, "获取统计信息失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, stats)  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
  
 "github.com/gin-gonic/gin"  
)  
  
type RoleController struct{}  
  
// List 角色记录列表  
// @Summary 获取角色记录列表  
// @Description 获取角色记录列表  
// @Tags 角色管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /role/list [get]  
// @Security ApiKeyAuth  
func (m \*RoleController) List(c \*gin.Context) {  
 req := new(request.RoleListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Role.List(c, req)  
 })  
}  
  
// Add 新建  
// @Summary 新建角色记录  
// @Description 新建角色记录  
// @Tags 角色管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.RoleAddReq true "添加角色记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /role/add [post]  
// @Security ApiKeyAuth  
func (m \*RoleController) Add(c \*gin.Context) {  
 req := new(request.RoleAddReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Role.Add(c, req)  
 })  
}  
  
// Update 更新记录  
// @Summary 更新角色记录  
// @Description 更新角色记录  
// @Tags 角色管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.RoleUpdateReq true "更新角色记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /role/update [post]  
// @Security ApiKeyAuth  
func (m \*RoleController) Update(c \*gin.Context) {  
 req := new(request.RoleUpdateReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Role.Update(c, req)  
 })  
}  
  
// Delete 删除记录  
// @Summary 删除角色记录  
// @Description 删除角色记录  
// @Tags 角色管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.RoleDeleteReq true "删除角色记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /role/delete [post]  
// @Security ApiKeyAuth  
func (m \*RoleController) Delete(c \*gin.Context) {  
 req := new(request.RoleDeleteReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Role.Delete(c, req)  
 })  
}  
  
// GetMenuList 获取菜单列表  
// @Summary 获取菜单列表  
// @Description 获取菜单列表  
// @Tags 角色管理  
// @Accept application/json  
// @Produce application/json  
// @Param roleId query int true "角色ID"  
// @Success 200 {object} response.ResponseBody  
// @Router /role/getmenulist [get]  
// @Security ApiKeyAuth  
func (m \*RoleController) GetMenuList(c \*gin.Context) {  
 req := new(request.RoleGetMenuListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Role.GetMenuList(c, req)  
 })  
}  
  
// GetApiList 获取接口列表  
// @Summary 获取接口列表  
// @Description 获取接口列表  
// @Tags 角色管理  
// @Accept application/json  
// @Produce application/json  
// @Param roleId query int true "角色ID"  
// @Success 200 {object} response.ResponseBody  
// @Router /role/getapilist [get]  
// @Security ApiKeyAuth  
func (m \*RoleController) GetApiList(c \*gin.Context) {  
 req := new(request.RoleGetApiListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Role.GetApiList(c, req)  
 })  
}  
  
// UpdateMenus 更新菜单  
// @Summary 更新菜单  
// @Description 更新菜单  
// @Tags 角色管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.RoleUpdateMenusReq true "更新菜单的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /role/updatemenus [post]  
// @Security ApiKeyAuth  
func (m \*RoleController) UpdateMenus(c \*gin.Context) {  
 req := new(request.RoleUpdateMenusReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Role.UpdateMenus(c, req)  
 })  
}  
  
// UpdateApis 更新接口  
// @Summary 更新接口  
// @Description 更新接口  
// @Tags 角色管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.RoleUpdateApisReq true "更新接口的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /role/updateapis [post]  
// @Security ApiKeyAuth  
func (m \*RoleController) UpdateApis(c \*gin.Context) {  
 req := new(request.RoleUpdateApisReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Role.UpdateApis(c, req)  
 })  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
  
 "github.com/gin-gonic/gin"  
)  
  
type GroupController struct{}  
  
// List 记录列表  
// @Summary 获取分组记录列表  
// @Description 获取分组记录列表  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /group/list [get]  
// @Security ApiKeyAuth  
func (m \*GroupController) List(c \*gin.Context) {  
 req := new(request.GroupListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.List(c, req)  
 })  
}  
  
// UserInGroup 在分组内的用户  
// @Summary 获取分组内用户  
// @Description 获取分组内用户  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Param groupId query int true "分组ID"  
// @Param nickname query string false "昵称"  
// @Success 200 {object} response.ResponseBody  
// @Router /group/useringroup [get]  
// @Security ApiKeyAuth  
func (m \*GroupController) UserInGroup(c \*gin.Context) {  
 req := new(request.UserInGroupReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.UserInGroup(c, req)  
 })  
}  
  
// UserNoInGroup 不在分组的用户  
// @Summary 不在分组的用户  
// @Description 不在分组的用户  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Param groupId query int true "分组ID"  
// @Param nickname query string false "昵称"  
// @Success 200 {object} response.ResponseBody  
// @Router /group/usernoingroup [get]  
// @Security ApiKeyAuth  
func (m \*GroupController) UserNoInGroup(c \*gin.Context) {  
 req := new(request.UserNoInGroupReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.UserNoInGroup(c, req)  
 })  
}  
  
// GetTree 接口树  
// @Summary 获取分组接口树  
// @Description 获取分组接口树  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /group/tree [get]  
// @Security ApiKeyAuth  
func (m \*GroupController) GetTree(c \*gin.Context) {  
 req := new(request.GroupListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.GetTree(c, req)  
 })  
}  
  
// Add 新建分组记录  
// @Summary 添加分组记录  
// @Description 添加分组记录  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.GroupAddReq true "添加用户记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /group/add [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) Add(c \*gin.Context) {  
 req := new(request.GroupAddReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.Add(c, req)  
 })  
}  
  
// Update 更新记录  
// @Summary 更新分组记录  
// @Description 更新分组记录  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.GroupUpdateReq true "更新用户记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /group/update [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) Update(c \*gin.Context) {  
 req := new(request.GroupUpdateReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.Update(c, req)  
 })  
}  
  
// Delete 删除记录  
// @Summary 删除分组记录  
// @Description 删除分组记录  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.GroupDeleteReq true "删除用户记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /group/delete [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) Delete(c \*gin.Context) {  
 req := new(request.GroupDeleteReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.Delete(c, req)  
 })  
}  
  
// AddUser 添加用户  
// @Summary 添加用户  
// @Description 添加用户  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.GroupAddUserReq true "添加用户记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /group/adduser [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) AddUser(c \*gin.Context) {  
 req := new(request.GroupAddUserReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.AddUser(c, req)  
 })  
}  
  
// RemoveUser 移除用户  
// @Summary 移除用户  
// @Description 移除用户  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.GroupRemoveUserReq true "移除用户记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /group/removeuser [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) RemoveUser(c \*gin.Context) {  
 req := new(request.GroupRemoveUserReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Group.RemoveUser(c, req)  
 })  
}  
  
// SyncDingTalkDepts 同步钉钉部门信息  
// @Summary 同步钉钉部门信息  
// @Description 同步钉钉部门信息  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /group/syncDingTalkDepts [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) SyncDingTalkDepts(c \*gin.Context) {  
 req := new(request.SyncDingTalkDeptsReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.DingTalk.SyncDingTalkDepts(c, req)  
 })  
}  
  
// SyncWeComDepts 同步企业微信部门信息  
// @Summary 同步企业微信部门信息  
// @Description 同步企业微信部门信息  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /group/syncWeComDepts [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) SyncWeComDepts(c \*gin.Context) {  
 req := new(request.SyncWeComDeptsReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.WeCom.SyncWeComDepts(c, req)  
 })  
}  
  
// SyncFeiShuDepts 同步飞书部门信息  
// @Summary 同步飞书部门信息  
// @Description 同步飞书部门信息  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /group/syncFeiShuDepts [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) SyncFeiShuDepts(c \*gin.Context) {  
 req := new(request.SyncFeiShuDeptsReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.FeiShu.SyncFeiShuDepts(c, req)  
 })  
}  
  
// SyncOpenLdapDepts 同步原ldap部门信息  
// @Summary 同步原ldap部门信息  
// @Description 同步原ldap部门信息  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /group/syncOpenLdapDepts [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) SyncOpenLdapDepts(c \*gin.Context) {  
 req := new(request.SyncOpenLdapDeptsReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.OpenLdap.SyncOpenLdapDepts(c, req)  
 })  
}  
  
// SyncSqlGroups 同步Sql中的分组信息到ldap  
// @Summary 同步Sql中的分组信息到ldap  
// @Description 同步Sql中的分组信息到ldap  
// @Tags 分组管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /group/syncSqlGroups [post]  
// @Security ApiKeyAuth  
func (m \*GroupController) SyncSqlGroups(c \*gin.Context) {  
 req := new(request.SyncSqlGrooupsReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Sql.SyncSqlGroups(c, req)  
 })  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
  
 "github.com/gin-gonic/gin"  
)  
  
type ApiController struct{}  
  
// List 记录列表  
// @Summary 获取API接口列表  
// Description: 获取API接口列表  
// @Tags 接口管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /api/list [get]  
// @Security ApiKeyAuth  
func (m \*ApiController) List(c \*gin.Context) {  
 req := new(request.ApiListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Api.List(c, req)  
 })  
}  
  
// GetTree 接口树  
// @Summary 获取API接口树  
// Description: 获取API接口树  
// @Tags 接口管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /api/tree [get]  
// @Security ApiKeyAuth  
func (m \*ApiController) GetTree(c \*gin.Context) {  
 req := new(request.ApiGetTreeReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Api.GetTree(c, req)  
 })  
}  
  
// Add 新建记录  
// @Summary 新建API接口  
// @Tags 接口管理  
// Description: 新建API接口  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.ApiAddReq true "新建API"  
// @Success 200 {object} response.ResponseBody  
// @Router /api/add [post]  
// @Security ApiKeyAuth  
func (m \*ApiController) Add(c \*gin.Context) {  
 req := new(request.ApiAddReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Api.Add(c, req)  
 })  
}  
  
// Update 更新记录  
// @Summary 更新API接口  
// @Tags 接口管理  
// Description: 更新API接口  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.ApiUpdateReq true "更新API"  
// @Success 200 {object} response.ResponseBody  
// @Router /api/update [post]  
// @Security ApiKeyAuth  
func (m \*ApiController) Update(c \*gin.Context) {  
 req := new(request.ApiUpdateReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Api.Update(c, req)  
 })  
}  
  
// Delete 删除记录  
// @Summary 删除API接口  
// @Tags 接口管理  
// Description: 删除API接口  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.ApiDeleteReq true "删除API"  
// @Success 200 {object} response.ResponseBody  
// @Router /api/delete [post]  
// @Security ApiKeyAuth  
func (m \*ApiController) Delete(c \*gin.Context) {  
 req := new(request.ApiDeleteReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Api.Delete(c, req)  
 })  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
  
 "github.com/gin-gonic/gin"  
)  
  
type OperationLogController struct{}  
  
// List 记录列表  
// @Summary 获取操作日志记录列表  
// Description: 获取操作日志记录列表  
// @Tags 操作日志管理  
// @Accept application/json  
// @Produce application/json  
// @Param username query string false "用户名"  
// @Param ip query string false "IP地址"  
// @Param path query string false "路径"  
// @Param method query string false "方法"  
// @Param status query int false "状态码"  
// @Param pageNum query int false "页码"  
// @Param pageSize query int false "每页数量"  
// @Success 200 {object} response.ResponseBody  
// @Router /log/operation/list [get]  
// @Security ApiKeyAuth  
func (m \*OperationLogController) List(c \*gin.Context) {  
 req := new(request.OperationLogListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.OperationLog.List(c, req)  
 })  
}  
  
// Delete 删除记录  
// @Summary 删除操作日志记录  
// Description: 删除操作日志记录  
// @Tags 操作日志管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.OperationLogDeleteReq true "删除日志的ID"  
// @Success 200 {object} response.ResponseBody  
// @Router /log/operation/delete [post]  
// @Security ApiKeyAuth  
func (m \*OperationLogController) Delete(c \*gin.Context) {  
 req := new(request.OperationLogDeleteReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.OperationLog.Delete(c, req)  
 })  
}  
  
// Clean 清空记录  
// @Summary 清空操作日志记录  
// Description: 清空操作日志记录  
// @Tags 操作日志管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /log/operation/clean [delete]  
// @Security ApiKeyAuth  
func (m \*OperationLogController) Clean(c \*gin.Context) {  
 req := new(request.OperationLogListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.OperationLog.Clean(c, req)  
 })  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
  
 "github.com/gin-gonic/gin"  
)  
  
type MenuController struct{}  
  
// GetTree 菜单树  
// @Summary 获取菜单树  
// @Tags 菜单管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /menu/tree [get]  
// @Security ApiKeyAuth  
func (m \*MenuController) GetTree(c \*gin.Context) {  
 req := new(request.MenuGetTreeReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Menu.GetTree(c, req)  
 })  
}  
  
// GetAccessTree GetUserMenuTreeByUserId 获取用户菜单树  
// @Summary 获取用户菜单树  
// @Tags 菜单管理  
// @Accept application/json  
// @Produce application/json  
// @Param id query int true "分组ID"  
// @Success 200 {object} response.ResponseBody  
// @Router /menu/access/tree [get]  
// @Security ApiKeyAuth  
func (m \*MenuController) GetAccessTree(c \*gin.Context) {  
 req := new(request.MenuGetAccessTreeReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Menu.GetAccessTree(c, req)  
 })  
}  
  
// Add 新建  
// @Summary 新建菜单  
// @Tags 菜单管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.MenuAddReq true "新建菜单"  
// @Success 200 {object} response.ResponseBody  
// @Router /menu/add [post]  
// @Security ApiKeyAuth  
func (m \*MenuController) Add(c \*gin.Context) {  
 req := new(request.MenuAddReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Menu.Add(c, req)  
 })  
}  
  
// Update 更新记录  
// @Summary 更新菜单  
// @Tags 菜单管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.MenuUpdateReq true "更新菜单"  
// @Success 200 {object} response.ResponseBody  
// @Router /menu/update [post]  
// @Security ApiKeyAuth  
func (m \*MenuController) Update(c \*gin.Context) {  
 req := new(request.MenuUpdateReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Menu.Update(c, req)  
 })  
}  
  
// Delete 删除记录  
// @Summary 删除菜单  
// @Tags 菜单管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.MenuDeleteReq true "删除菜单"  
// @Success 200 {object} response.ResponseBody  
// @Router /menu/delete [post]  
// @Security ApiKeyAuth  
func (m \*MenuController) Delete(c \*gin.Context) {  
 req := new(request.MenuDeleteReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Menu.Delete(c, req)  
 })  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
  
 "github.com/gin-gonic/gin"  
)  
  
type UserController struct{}  
  
// Add 添加用户记录  
// @Summary 添加用户记录  
// @Description 添加用户记录  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.UserAddReq true "添加用户记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/add [post]  
// @Security ApiKeyAuth  
func (m \*UserController) Add(c \*gin.Context) {  
 req := new(request.UserAddReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.User.Add(c, req)  
 })  
}  
  
// Update 更新用户记录  
// @Summary 更新用户记录  
// @Description 添加用户记录  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.UserUpdateReq true "更改用户记录的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/update [post]  
// @Security ApiKeyAuth  
func (m \*UserController) Update(c \*gin.Context) {  
 req := new(request.UserUpdateReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.User.Update(c, req)  
 })  
}  
  
// List 记录列表  
// @Summary 获取所有用户记录列表  
// @Description 获取所有用户记录列表  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /user/list [get]  
// @Security ApiKeyAuth  
func (m \*UserController) List(c \*gin.Context) {  
 req := new(request.UserListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.User.List(c, req)  
 })  
}  
  
// Delete 删除用户记录  
// @Summary 删除用户记录  
// @Description 删除用户记录  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.UserDeleteReq true "删除用户记录的结构体ID"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/delete [post]  
// @Security ApiKeyAuth  
func (m UserController) Delete(c \*gin.Context) {  
 req := new(request.UserDeleteReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.User.Delete(c, req)  
 })  
}  
  
// ChangePwd 更新密码  
// @Summary 更新密码  
// @Description 更新密码  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.UserChangePwdReq true "更改用户密码的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/changePwd [post]  
// @Security ApiKeyAuth  
func (m UserController) ChangePwd(c \*gin.Context) {  
 req := new(request.UserChangePwdReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.User.ChangePwd(c, req)  
 })  
}  
  
// ChangeUserStatus 更改用户状态  
// @Summary 更改用户状态  
// @Description 更改用户状态  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.UserChangeUserStatusReq true "更改用户状态的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/changeUserStatus [post]  
// @Security ApiKeyAuth  
func (m UserController) ChangeUserStatus(c \*gin.Context) {  
 req := new(request.UserChangeUserStatusReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.User.ChangeUserStatus(c, req)  
 })  
}  
  
// GetUserInfo 获取当前登录用户信息  
// @Summary 获取当前登录用户信息  
// @Description 获取当前登录用户信息  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /user/info [get]  
// @Security ApiKeyAuth  
func (uc UserController) GetUserInfo(c \*gin.Context) {  
 req := new(request.UserGetUserInfoReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.User.GetUserInfo(c, req)  
 })  
}  
  
// SyncDingTalkUsers 同步钉钉用户信息  
// @Summary 同步钉钉用户信息  
// @Description 同步钉钉用户信息  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.SyncDingUserReq true "同步钉钉用户信息"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/syncDingTalkUsers [post]  
// @Security ApiKeyAuth  
func (uc UserController) SyncDingTalkUsers(c \*gin.Context) {  
 req := new(request.SyncDingUserReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.DingTalk.SyncDingTalkUsers(c, req)  
 })  
}  
  
// SyncWeComUsers 同步企业微信用户信息  
// @Summary 同步企业微信用户信息  
// @Description 同步企业微信用户信息  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.SyncWeComUserReq true "同步企业微信用户信息"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/syncWeComUsers [post]  
// @Security ApiKeyAuth  
func (uc UserController) SyncWeComUsers(c \*gin.Context) {  
 req := new(request.SyncWeComUserReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.WeCom.SyncWeComUsers(c, req)  
 })  
}  
  
// SyncFeiShuUsers 同步飞书用户信息  
// @Summary 同步飞书用户信息  
// @Description 同步飞书用户信息  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.SyncFeiShuUserReq true "同步飞书用户信息"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/syncFeiShuUsers [post]  
// @Security ApiKeyAuth  
func (uc UserController) SyncFeiShuUsers(c \*gin.Context) {  
 req := new(request.SyncFeiShuUserReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.FeiShu.SyncFeiShuUsers(c, req)  
 })  
}  
  
// SyncOpenLdapUsers 同步ldap用户信息  
// @Summary 同步ldap用户信息  
// @Description 同步ldap用户信息  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.SyncOpenLdapUserReq true "同步ldap用户信息"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/syncOpenLdapUsers [post]  
// @Security ApiKeyAuth  
func (uc UserController) SyncOpenLdapUsers(c \*gin.Context) {  
 req := new(request.SyncOpenLdapUserReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.OpenLdap.SyncOpenLdapUsers(c, req)  
 })  
}  
  
// SyncSqlUsers 同步sql用户信息到ldap  
// @Summary 同步sql用户信息到ldap  
// @Description 同步sql用户信息到ldap  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.SyncSqlUserReq true "更改用户状态的结构体"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/syncSqlUsers [post]  
// @Security ApiKeyAuth  
func (uc UserController) SyncSqlUsers(c \*gin.Context) {  
 req := new(request.SyncSqlUserReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Sql.SyncSqlUsers(c, req)  
 })  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/gin-gonic/gin"  
)  
  
type LDAPMonitorController struct{}  
  
var LDAPMonitor = new(LDAPMonitorController)  
  
// GetStatus 获取LDAP连接状态  
// @Summary 获取LDAP连接状态  
// @Description 获取LDAP服务器连接状态信息  
// @Tags LDAP监控  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /ldap-monitor/status [get]  
func (m \*LDAPMonitorController) GetStatus(c \*gin.Context) {  
 Run(c, nil, func() (interface{}, interface{}) {  
 return logic.LDAPMonitor.GetStatus(c)  
 })  
}  
  
// CheckConnection 检查LDAP连接  
// @Summary 检查LDAP连接  
// @Description 手动触发LDAP连接检查  
// @Tags LDAP监控  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /ldap-monitor/check [post]  
func (m \*LDAPMonitorController) CheckConnection(c \*gin.Context) {  
 Run(c, nil, func() (interface{}, interface{}) {  
 return logic.LDAPMonitor.CheckConnection(c)  
 })  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
 "github.com/gin-gonic/gin"  
)  
  
type UserImportController struct{}  
  
var UserImport = new(UserImportController)  
  
// ImportUsers 批量导入用户  
// @Summary 批量导入用户  
// @Description 批量导入用户到LDAP  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.UserImportReq true "用户导入信息"  
// @Success 200 {object} response.ResponseBody  
// @Router /user/import [post]  
func (m \*UserImportController) ImportUsers(c \*gin.Context) {  
 req := new(request.UserImportReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.UserImport.ImportUsers(c, req)  
 })  
}  
  
// ExportUsers 导出用户  
// @Summary 导出用户  
// @Description 导出LDAP用户信息  
// @Tags 用户管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /user/export [get]  
func (m \*UserImportController) ExportUsers(c \*gin.Context) {  
 Run(c, nil, func() (interface{}, interface{}) {  
 return logic.UserImport.ExportUsers(c)  
 })  
}

package controller  
  
import (  
 "strconv"  
  
 "github.com/eryajf/go-ldap-admin/model"  
 "github.com/eryajf/go-ldap-admin/public/response"  
 "github.com/eryajf/go-ldap-admin/service/isql"  
 "github.com/gin-gonic/gin"  
)  
  
type ichInheritorController struct{}  
  
var ICHInheritor = new(ichInheritorController)  
  
// Add 添加非遗传承人  
func (m \*ichInheritorController) Add(c \*gin.Context) {  
 var inheritor model.ICHInheritor  
 err := c.ShouldBindJSON(&inheritor)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 err = isql.ICHInheritor.Add(&inheritor)  
 if err != nil {  
 response.Failed(c, "添加失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, "添加成功")  
}  
  
// Update 更新非遗传承人信息  
func (m \*ichInheritorController) Update(c \*gin.Context) {  
 var inheritor model.ICHInheritor  
 err := c.ShouldBindJSON(&inheritor)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 err = isql.ICHInheritor.Update(&inheritor)  
 if err != nil {  
 response.Failed(c, "更新失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, "更新成功")  
}  
  
// Delete 删除非遗传承人  
func (m \*ichInheritorController) Delete(c \*gin.Context) {  
 id := c.Param("id")  
 idUint, err := strconv.ParseUint(id, 10, 32)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 err = isql.ICHInheritor.Delete(uint(idUint))  
 if err != nil {  
 response.Failed(c, "删除失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, "删除成功")  
}  
  
// GetByID 根据ID获取非遗传承人信息  
func (m \*ichInheritorController) GetByID(c \*gin.Context) {  
 id := c.Param("id")  
 idUint, err := strconv.ParseUint(id, 10, 32)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 inheritor, err := isql.ICHInheritor.GetByID(uint(idUint))  
 if err != nil {  
 response.Failed(c, "获取失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, inheritor)  
}  
  
// List 获取非遗传承人列表  
func (m \*ichInheritorController) List(c \*gin.Context) {  
 page, \_ := strconv.Atoi(c.DefaultQuery("page", "1"))  
 size, \_ := strconv.Atoi(c.DefaultQuery("size", "10"))  
 query := c.DefaultQuery("query", "")  
  
 inheritors, total, err := isql.ICHInheritor.List(page, size, query)  
 if err != nil {  
 response.Failed(c, "获取失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, gin.H{  
 "items": inheritors,  
 "total": total,  
 })  
}  
  
// UpdateStatus 更新传承人状态  
func (m \*ichInheritorController) UpdateStatus(c \*gin.Context) {  
 id := c.Param("id")  
 idUint, err := strconv.ParseUint(id, 10, 32)  
 if err != nil {  
 response.Failed(c, "参数验证失败")  
 return  
 }  
  
 status, err := strconv.Atoi(c.DefaultQuery("status", "1"))  
 if err != nil {  
 response.Failed(c, "状态参数错误")  
 return  
 }  
  
 err = isql.ICHInheritor.UpdateStatus(uint(idUint), status)  
 if err != nil {  
 response.Failed(c, "更新状态失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, "更新状态成功")  
}  
  
// GetByProjectLevel 根据项目级别获取传承人列表  
func (m \*ichInheritorController) GetByProjectLevel(c \*gin.Context) {  
 level := c.Query("level")  
 if level == "" {  
 response.Failed(c, "项目级别不能为空")  
 return  
 }  
  
 inheritors, err := isql.ICHInheritor.GetByProjectLevel(level)  
 if err != nil {  
 response.Failed(c, "获取失败: "+err.Error())  
 return  
 }  
  
 response.Success(c, inheritors)  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
  
 "github.com/gin-gonic/gin"  
)  
  
type BaseController struct{}  
  
// SendCode 给用户邮箱发送验证码  
// @Summary 发送验证码  
// @Description 向指定邮箱发送验证码  
// @Tags 基础管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.BaseSendCodeReq true "发送验证码请求数据"  
// @Success 200 {object} response.ResponseBody  
// @Router /base/sendcode [post]  
func (m \*BaseController) SendCode(c \*gin.Context) {  
 req := new(request.BaseSendCodeReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Base.SendCode(c, req)  
 })  
}  
  
// ChangePwd 用户通过邮箱修改密码  
// @Summary 用户通过邮箱修改密码  
// @Description 使用邮箱验证码修改密码  
// @Tags 基础管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.BaseChangePwdReq true "发送验证码请求数据"  
// @Success 200 {object} response.ResponseBody  
// @Router /base/changePwd [post]  
func (m \*BaseController) ChangePwd(c \*gin.Context) {  
 req := new(request.BaseChangePwdReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Base.ChangePwd(c, req)  
 })  
}  
  
// Dashboard 系统首页展示数据  
// @Summary 获取仪表盘数据  
// @Description 获取系统仪表盘概览数据  
// @Tags 基础管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /base/dashboard [get]  
func (m \*BaseController) Dashboard(c \*gin.Context) {  
 req := new(request.BaseDashboardReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Base.Dashboard(c, req)  
 })  
}  
  
// EncryptPasswd 密码加密  
// @Summary 密码加密  
// @Description 将明文密码加密  
// @Tags 基础管理  
// @Accept application/json  
// @Produce application/json  
// @Param passwd query string true "需要加密的明文密码"  
// @Success 200 {object} response.ResponseBody  
// @Router /base/encryptpwd [get]  
func (m \*BaseController) EncryptPasswd(c \*gin.Context) {  
 req := new(request.EncryptPasswdReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Base.EncryptPasswd(c, req)  
 })  
}  
  
// DecryptPasswd 密码解密为明文  
// @Summary 密码解密  
// @Description 将加密后的密码解密为明文  
// @Tags 基础管理  
// @Accept application/json  
// @Produce application/json  
// @Param passwd query string true "需要解密的加密密码"  
// @Success 200 {object} response.ResponseBody  
// @Router /base/decryptpwd [get]  
func (m \*BaseController) DecryptPasswd(c \*gin.Context) {  
 req := new(request.DecryptPasswdReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.Base.DecryptPasswd(c, req)  
 })  
}

package controller  
  
import (  
 "github.com/eryajf/go-ldap-admin/logic"  
 "github.com/eryajf/go-ldap-admin/model/request"  
  
 "github.com/gin-gonic/gin"  
)  
  
type FieldRelationController struct{}  
  
// List 记录列表  
// @Summary 获字段关系管理列表  
// Description: 获字段关系管理列表  
// @Tags 字段关系管理  
// @Accept application/json  
// @Produce application/json  
// @Success 200 {object} response.ResponseBody  
// @Router /fieldrelation/list [get]  
// @Security ApiKeyAuth  
func (m \*FieldRelationController) List(c \*gin.Context) {  
 req := new(request.FieldRelationListReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.FieldRelation.List(c, req)  
 })  
}  
  
// Add 新建记录  
// @Summary 新建字段关系管理记录  
// Description: 新建字段关系管理记录  
// @Tags 字段关系管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.FieldRelationAddReq true "新建字段关系管理记录"  
// @Success 200 {object} response.ResponseBody  
// @Router /fieldrelation/add [post]  
// @Security ApiKeyAuth  
func (m \*FieldRelationController) Add(c \*gin.Context) {  
 req := new(request.FieldRelationAddReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.FieldRelation.Add(c, req)  
 })  
}  
  
// Update 更新记录  
// @Summary 更新字段关系管理记录  
// Description: 更新字段关系管理记录  
// @Tags 字段关系管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.FieldRelationUpdateReq true "更新字段关系管理记录"  
// @Success 200 {object} response.ResponseBody  
// @Router /fieldrelation/update [post]  
// @Security ApiKeyAuth  
func (m \*FieldRelationController) Update(c \*gin.Context) {  
 req := new(request.FieldRelationUpdateReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.FieldRelation.Update(c, req)  
 })  
}  
  
// Delete 删除记录  
// @Summary 删除字段关系管理记录  
// Description: 删除字段关系管理记录  
// @Tags 字段关系管理  
// @Accept application/json  
// @Produce application/json  
// @Param data body request.FieldRelationDeleteReq true "删除字段关系管理记录"  
// @Success 200 {object} response.ResponseBody  
// @Router /fieldrelation/delete [post]  
// @Security ApiKeyAuth  
func (m \*FieldRelationController) Delete(c \*gin.Context) {  
 req := new(request.FieldRelationDeleteReq)  
 Run(c, req, func() (interface{}, interface{}) {  
 return logic.FieldRelation.Delete(c, req)  
 })  
}

package model  
  
import "gorm.io/gorm"  
  
// HeritageAchievement 非遗传承成果信息表  
type HeritageAchievement struct {  
 gorm.Model  
 MasterID uint `gorm:"type:int;comment:'传承人ID'" json:"masterId"` // 关联的传承人ID  
 Name string `gorm:"type:varchar(100);comment:'成果名称'" json:"name"` // 成果名称  
 Type string `gorm:"type:varchar(50);comment:'成果类型'" json:"type"` // 成果类型(作品/专利/获奖)  
 Category string `gorm:"type:varchar(50);comment:'作品类别'" json:"category"` // 作品类别(实物/非实物)  
 Description string `gorm:"type:text;comment:'成果描述'" json:"description"` // 成果描述  
 CreationDate string `gorm:"type:varchar(20);comment:'创作日期'" json:"creationDate"` // 创作/获得日期  
  
 // 专利相关信息  
 PatentNo string `gorm:"type:varchar(50);comment:'专利号'" json:"patentNo"` // 专利号  
 PatentType string `gorm:"type:varchar(50);comment:'专利类型'" json:"patentType"` // 专利类型  
 PatentStatus string `gorm:"type:varchar(50);comment:'专利状态'" json:"patentStatus"` // 专利状态  
 GrantDate string `gorm:"type:varchar(20);comment:'授权日期'" json:"grantDate"` // 授权日期  
  
 // 获奖相关信息  
 AwardName string `gorm:"type:varchar(100);comment:'奖项名称'" json:"awardName"` // 奖项名称  
 AwardLevel string `gorm:"type:varchar(50);comment:'获奖级别'" json:"awardLevel"` // 获奖级别(国家级/省级/市级)  
 AwardDate string `gorm:"type:varchar(20);comment:'获奖日期'" json:"awardDate"` // 获奖日期  
 AwardingBody string `gorm:"type:varchar(100);comment:'颁奖单位'" json:"awardingBody"` // 颁奖单位  
  
 // 作品相关信息  
 Material string `gorm:"type:varchar(255);comment:'材料工艺'" json:"material"` // 材料工艺  
 Size string `gorm:"type:varchar(50);comment:'作品规格'" json:"size"` // 作品规格  
 Location string `gorm:"type:varchar(255);comment:'收藏地点'" json:"location"` // 收藏/展示地点  
  
 // 通用字段  
 MediaFiles string `gorm:"type:text;comment:'相关媒体文件'" json:"mediaFiles"` // 相关图片/视频等媒体文件  
 Recognition string `gorm:"type:text;comment:'社会认可度'" json:"recognition"` // 社会认可度/影响力  
 Notes string `gorm:"type:text;comment:'备注'" json:"notes"` // 其他说明  
}

package model  
  
import "gorm.io/gorm"  
  
// PasswordPolicy 密码策略表  
type PasswordPolicy struct {  
 gorm.Model  
 MinLength int `gorm:"type:int;default:6;comment:'最小长度'" json:"minLength"` // 最小长度  
 RequireUppercase bool `gorm:"type:tinyint(1);default:0;comment:'要求大写字母'" json:"requireUppercase"` // 要求大写字母  
 RequireLowercase bool `gorm:"type:tinyint(1);default:0;comment:'要求小写字母'" json:"requireLowercase"` // 要求小写字母  
 RequireNumber bool `gorm:"type:tinyint(1);default:0;comment:'要求数字'" json:"requireNumber"` // 要求数字  
 RequireSpecialChar bool `gorm:"type:tinyint(1);default:0;comment:'要求特殊字符'" json:"requireSpecialChar"` // 要求特殊字符  
 MaxAge int `gorm:"type:int;default:0;comment:'密码最大有效期(天)'" json:"maxAge"` // 密码最大有效期(天)  
 HistoryCount int `gorm:"type:int;default:0;comment:'密码历史记录数'" json:"historyCount"` // 密码历史记录数  
 LockoutThreshold int `gorm:"type:int;default:0;comment:'锁定阈值'" json:"lockoutThreshold"` // 锁定阈值  
 LockoutDuration int `gorm:"type:int;default:0;comment:'锁定时间(分钟)'" json:"lockoutDuration"` // 锁定时间(分钟)  
}

package model  
  
import (  
 "gorm.io/gorm"  
)  
  
type OperationLog struct {  
 gorm.Model  
 Username string `gorm:"type:varchar(20);comment:'用户登录名'" json:"username"`  
 Ip string `gorm:"type:varchar(20);comment:'Ip地址'" json:"ip"`  
 IpLocation string `gorm:"type:varchar(20);comment:'Ip所在地'" json:"ipLocation"`  
 Method string `gorm:"type:varchar(20);comment:'请求方式'" json:"method"`  
 Path string `gorm:"type:varchar(100);comment:'访问路径'" json:"path"`  
 Remark string `gorm:"type:varchar(100);comment:'备注'" json:"remark"`  
 Status int `gorm:"type:int(4);comment:'响应状态码'" json:"status"`  
 StartTime string `gorm:"type:varchar(2048);comment:'发起时间'" json:"startTime"`  
 TimeCost int64 `gorm:"type:int(6);comment:'请求耗时(ms)'" json:"timeCost"`  
 UserAgent string `gorm:"type:varchar(2048);comment:'浏览器标识'" json:"userAgent"`  
}

package model  
  
import "gorm.io/gorm"  
  
// HeritageApprentice 非遗传承人学徒信息表  
type HeritageApprentice struct {  
 gorm.Model  
 MasterID uint `gorm:"type:int;comment:'传承人ID'" json:"masterId"` // 关联的传承人ID  
 Name string `gorm:"type:varchar(50);comment:'姓名'" json:"name"` // 姓名  
 Gender string `gorm:"type:varchar(10);comment:'性别'" json:"gender"` // 性别  
 Birthday string `gorm:"type:varchar(20);comment:'出生日期'" json:"birthday"` // 出生日期  
 IDCard string `gorm:"type:varchar(18);comment:'身份证号'" json:"-"` // 身份证号(加密存储)  
 Phone string `gorm:"type:varchar(20);comment:'联系电话'" json:"phone"` // 联系电话  
 Email string `gorm:"type:varchar(100);comment:'电子邮箱'" json:"email"` // 电子邮箱  
 Address string `gorm:"type:varchar(255);comment:'联系地址'" json:"address"` // 联系地址  
 Education string `gorm:"type:varchar(50);comment:'学历'" json:"education"` // 学历  
 StartDate string `gorm:"type:varchar(20);comment:'开始学习日期'" json:"startDate"` // 开始学习日期  
 EndDate string `gorm:"type:varchar(20);comment:'结束学习日期'" json:"endDate"` // 结束学习日期  
 LearningPlan string `gorm:"type:text;comment:'学习计划'" json:"learningPlan"` // 学习计划  
 Progress string `gorm:"type:text;comment:'学习进度'" json:"progress"` // 学习进度  
 Evaluation string `gorm:"type:text;comment:'考核评价'" json:"evaluation"` // 考核评价  
 Certificate string `gorm:"type:varchar(50);comment:'结业证书编号'" json:"certificate"` // 结业证书编号  
 Status int `gorm:"type:int;default:1;comment:'状态'" json:"status"` // 状态(1:在学/2:结业/3:终止)  
 MentorshipType string `gorm:"type:varchar(50);comment:'师承类型'" json:"mentorshipType"` // 师承类型(全职/非全职)  
 SkillLevel string `gorm:"type:varchar(50);comment:'技艺水平'" json:"skillLevel"` // 技艺水平评估  
 Notes string `gorm:"type:text;comment:'备注'" json:"notes"` // 其他说明  
}

package model  
  
import "gorm.io/gorm"  
  
type User struct {  
 gorm.Model  
 Username string `gorm:"type:varchar(50);not null;unique;comment:'用户名'" json:"username"` // 用户名  
 Password string `gorm:"size:255;not null;comment:'用户密码'" json:"password"` // 用户密码  
 Nickname string `gorm:"type:varchar(50);comment:'中文名'" json:"nickname"` // 昵称  
 GivenName string `gorm:"type:varchar(50);comment:'花名'" json:"givenName"` // 花名，如果有的话，没有的话用昵称占位  
 Mail string `gorm:"type:varchar(100);comment:'邮箱'" json:"mail"` // 邮箱  
 JobNumber string `gorm:"type:varchar(20);comment:'工号'" json:"jobNumber"` // 工号  
 Mobile string `gorm:"type:varchar(15);not null;unique;comment:'手机号'" json:"mobile"` // 手机号  
 Avatar string `gorm:"type:varchar(255);comment:'头像'" json:"avatar"` // 头像  
 PostalAddress string `gorm:"type:varchar(255);comment:'地址'" json:"postalAddress"` // 地址  
 Departments string `gorm:"type:varchar(512);comment:'部门'" json:"departments"` // 部门  
 Position string `gorm:"type:varchar(128);comment:'职位'" json:"position"` // 职位  
 Introduction string `gorm:"type:varchar(255);comment:'个人简介'" json:"introduction"` // 个人简介  
 Status uint `gorm:"type:tinyint(1);default:1;comment:'状态:1在职, 2离职'" json:"status"` // 状态  
 Creator string `gorm:"type:varchar(20);;comment:'创建者'" json:"creator"` // 创建者  
 Source string `gorm:"type:varchar(50);comment:'用户来源：dingTalk、wecom、feishu、ldap、platform'" json:"source"` // 来源  
 DepartmentId string `gorm:"type:varchar(100);not null;comment:'部门id'" json:"departmentId"` // 部门id  
 Roles []\*Role `gorm:"many2many:user\_roles" json:"roles"` // 角色  
 SourceUserId string `gorm:"type:varchar(100);not null;comment:'第三方用户id'" json:"sourceUserId"` // 第三方用户id  
 SourceUnionId string `gorm:"type:varchar(100);not null;comment:'第三方唯一unionId'" json:"sourceUnionId"` // 第三方唯一unionId  
 UserDN string `gorm:"type:varchar(255);not null;comment:'用户dn'" json:"userDn"` // 用户在ldap的dn  
 SyncState uint `gorm:"type:tinyint(1);default:1;comment:'同步状态:1已同步, 2未同步'" json:"syncState"` // 数据到ldap的同步状态  
}  
  
func (u \*User) SetUserName(userName string) {  
 u.Username = userName  
}  
  
func (u \*User) SetNickName(nickName string) {  
 u.Nickname = nickName  
}  
  
func (u \*User) SetGivenName(givenName string) {  
 u.GivenName = givenName  
}  
  
func (u \*User) SetMail(mail string) {  
 u.Mail = mail  
}  
  
func (u \*User) SetJobNumber(jobNum string) {  
 u.JobNumber = jobNum  
}  
  
func (u \*User) SetMobile(mobile string) {  
 u.Mobile = mobile  
}  
  
func (u \*User) SetAvatar(avatar string) {  
 u.Avatar = avatar  
}  
  
func (u \*User) SetPostalAddress(address string) {  
 u.PostalAddress = address  
}  
  
func (u \*User) SetPosition(position string) {  
 u.Position = position  
}  
  
func (u \*User) SetIntroduction(desc string) {  
 u.Introduction = desc  
}  
  
func (u \*User) SetSourceUserId(sourceUserId string) {  
 u.SourceUserId = sourceUserId  
}  
  
func (u \*User) SetSourceUnionId(sourceUnionId string) {  
 u.SourceUnionId = sourceUnionId  
}

package model  
  
// 角色权限规则  
type RoleCasbin struct {  
 Keyword string `json:"keyword"` // 角色关键字  
 Path string `json:"path"` // 访问路径  
 Method string `json:"method"` // 请求方式  
}

package model  
  
import "gorm.io/gorm"  
  
// HeritageEvent 非遗传承活动记录表  
type HeritageEvent struct {  
 gorm.Model  
 MasterID uint `gorm:"type:int;comment:'传承人ID'" json:"masterId"` // 传承人ID  
 EventName string `gorm:"type:varchar(100);comment:'活动名称'" json:"eventName"` // 活动名称  
 EventType string `gorm:"type:varchar(50);comment:'活动类型'" json:"eventType"` // 活动类型(展演/培训/讲座等)  
 StartTime string `gorm:"type:varchar(20);comment:'开始时间'" json:"startTime"` // 开始时间  
 EndTime string `gorm:"type:varchar(20);comment:'结束时间'" json:"endTime"` // 结束时间  
 Location string `gorm:"type:varchar(255);comment:'活动地点'" json:"location"` // 活动地点  
 Participants int `gorm:"type:int;comment:'参与人数'" json:"participants"` // 参与人数  
 Budget float64 `gorm:"type:decimal(10,2);comment:'活动预算'" json:"budget"` // 活动预算  
 Expense float64 `gorm:"type:decimal(10,2);comment:'实际支出'" json:"expense"` // 实际支出  
 Status int `gorm:"type:int;default:0;comment:'状态'" json:"status"` // 状态(0:筹备中/1:进行中/2:已结束)  
 Summary string `gorm:"type:text;comment:'活动总结'" json:"summary"` // 活动总结  
 Attachments string `gorm:"type:text;comment:'附件列表'" json:"attachments"` // 附件列表(JSON格式)  
 Organizer string `gorm:"type:varchar(100);comment:'主办方'" json:"organizer"` // 主办方  
 Collaborators string `gorm:"type:text;comment:'协办方'" json:"collaborators"` // 协办方  
 Sponsors string `gorm:"type:text;comment:'赞助方'" json:"sponsors"` // 赞助方  
 Target string `gorm:"type:varchar(255);comment:'活动对象'" json:"target"` // 活动对象  
 Scale string `gorm:"type:varchar(50);comment:'活动规模'" json:"scale"` // 活动规模  
 Effect string `gorm:"type:text;comment:'活动效果'" json:"effect"` // 活动效果  
 MediaReports string `gorm:"type:text;comment:'媒体报道'" json:"mediaReports"` // 媒体报道  
 Pictures string `gorm:"type:text;comment:'活动照片'" json:"pictures"` // 活动照片  
 Videos string `gorm:"type:text;comment:'活动视频'" json:"videos"` // 活动视频  
}

package model  
  
import "gorm.io/gorm"  
  
// LDAPMonitor LDAP监控状态表  
type LDAPMonitor struct {  
 gorm.Model  
 ConnectionStatus bool `gorm:"type:tinyint(1);default:0;comment:'连接状态'" json:"connectionStatus"` // 连接状态  
 ResponseTime int64 `gorm:"type:bigint;comment:'响应时间(ms)'" json:"responseTime"` // 响应时间(ms)  
 LastCheckTime string `gorm:"type:varchar(20);comment:'最后检查时间'" json:"lastCheckTime"` // 最后检查时间  
 ErrorCount int `gorm:"type:int;default:0;comment:'错误次数'" json:"errorCount"` // 错误次数  
 LastError string `gorm:"type:varchar(255);comment:'最后错误信息'" json:"lastError"` // 最后错误信息  
 TotalUsers int `gorm:"type:int;default:0;comment:'总用户数'" json:"totalUsers"` // 总用户数  
 TotalGroups int `gorm:"type:int;default:0;comment:'总组数'" json:"totalGroups"` // 总组数  
 CPUUsage float64 `gorm:"type:decimal(5,2);comment:'CPU使用率(%)'" json:"cpuUsage"` // CPU使用率(%)  
 MemoryUsage float64 `gorm:"type:decimal(5,2);comment:'内存使用率(%)'" json:"memoryUsage"` // 内存使用率(%)  
 DiskUsage float64 `gorm:"type:decimal(5,2);comment:'磁盘使用率(%)'" json:"diskUsage"` // 磁盘使用率(%)  
 QPS float64 `gorm:"type:decimal(10,2);comment:'每秒查询数'" json:"qps"` // 每秒查询数  
 AvgResponseTime float64 `gorm:"type:decimal(10,2);comment:'平均响应时间(ms)'" json:"avgResponseTime"` // 平均响应时间(ms)  
}

package model  
  
import "gorm.io/gorm"  
  
type Role struct {  
 gorm.Model  
 Name string `gorm:"type:varchar(20);not null;unique" json:"name"`  
 Keyword string `gorm:"type:varchar(20);not null;unique" json:"keyword"`  
 Remark string `gorm:"type:varchar(100);comment:'备注'" json:"remark"`  
 Status uint `gorm:"type:tinyint(1);default:1;comment:'1正常, 2禁用'" json:"status"`  
 Sort uint `gorm:"type:int(3);default:999;comment:'角色排序(排序越大权限越低, 不能查看比自己序号小的角色, 不能编辑同序号用户权限, 排序为1表示超级管理员)'" json:"sort"`  
 Creator string `gorm:"type:varchar(20);" json:"creator"`  
 Users []\*User `gorm:"many2many:user\_roles" json:"users"`  
 Menus []\*Menu `gorm:"many2many:role\_menus;" json:"menus"` // 角色菜单多对多关系  
}

package model  
  
import (  
 "gorm.io/gorm"  
)  
  
type Group struct {  
 gorm.Model  
 GroupName string `gorm:"type:varchar(128);comment:'分组名称'" json:"groupName"`  
 Remark string `gorm:"type:varchar(128);comment:'分组中文说明'" json:"remark"`  
 Creator string `gorm:"type:varchar(20);comment:'创建人'" json:"creator"`  
 GroupType string `gorm:"type:varchar(20);comment:'分组类型：cn、ou'" json:"groupType"`  
 Users []\*User `gorm:"many2many:group\_users" json:"users"`  
 ParentId uint `gorm:"default:0;comment:'父组编号(编号为0时表示根组)'" json:"parentId"`  
 SourceDeptId string `gorm:"type:varchar(100);comment:'部门编号'" json:"sourceDeptId"`  
 Source string `gorm:"type:varchar(20);comment:'来源：dingTalk、weCom、ldap、platform'" json:"source"`  
 SourceDeptParentId string `gorm:"type:varchar(100);comment:'父部门编号'" json:"sourceDeptParentId"`  
 SourceUserNum int `gorm:"default:0;comment:'部门下的用户数量，从第三方获取的数据'" json:"source\_user\_num"`  
 Children []\*Group `gorm:"-" json:"children"`  
 GroupDN string `gorm:"type:varchar(255);not null;comment:'分组dn'" json:"groupDn"` // 分组在ldap的dn  
 SyncState uint `gorm:"type:tinyint(1);default:1;comment:'同步状态:1已同步, 2未同步'" json:"syncState"` // 数据到ldap的同步状态  
}  
  
func (g \*Group) SetGroupName(groupName string) {  
 g.GroupName = groupName  
}  
  
func (g \*Group) SetRemark(remark string) {  
 g.Remark = remark  
}  
  
func (g \*Group) SetSourceDeptId(sourceDeptId string) {  
 g.SourceDeptId = sourceDeptId  
}  
  
func (g \*Group) SetSourceDeptParentId(sourceDeptParentId string) {  
 g.SourceDeptParentId = sourceDeptParentId  
}

package model  
  
import "gorm.io/gorm"  
  
// HeritageMaster 非遗传承人信息表  
type HeritageMaster struct {  
 gorm.Model  
 UserID uint `gorm:"type:int;comment:'用户ID'" json:"userId"` // 关联用户ID  
 Name string `gorm:"type:varchar(50);comment:'姓名'" json:"name"` // 姓名  
 Gender string `gorm:"type:varchar(10);comment:'性别'" json:"gender"` // 性别  
 Birthday string `gorm:"type:varchar(20);comment:'出生日期'" json:"birthday"` // 出生日期  
 IDCard string `gorm:"type:varchar(18);comment:'身份证号'" json:"-"` // 身份证号(加密存储)  
 Phone string `gorm:"type:varchar(20);comment:'联系电话'" json:"phone"` // 联系电话  
 Email string `gorm:"type:varchar(100);comment:'电子邮箱'" json:"email"` // 电子邮箱  
 Address string `gorm:"type:varchar(255);comment:'联系地址'" json:"address"` // 联系地址  
 ProjectName string `gorm:"type:varchar(100);comment:'非遗项目名称'" json:"projectName"` // 非遗项目名称  
 ProjectLevel string `gorm:"type:varchar(50);comment:'项目级别'" json:"projectLevel"` // 项目级别(国家级/省级/市级/县级)  
 ProjectType string `gorm:"type:varchar(50);comment:'项目类别'" json:"projectType"` // 项目类别  
 CertificateNo string `gorm:"type:varchar(50);comment:'证书编号'" json:"certificateNo"` // 证书编号  
 CertificateDate string `gorm:"type:varchar(20);comment:'认定日期'" json:"certificateDate"` // 认定日期  
 Status int `gorm:"type:int;default:1;comment:'状态'" json:"status"` // 状态(1:在传/2:停传/3:终止)  
 Description string `gorm:"type:text;comment:'传承人简介'" json:"description"` // 传承人简介  
 Nationality string `gorm:"type:varchar(50);comment:'民族'" json:"nationality"` // 民族  
 Education string `gorm:"type:varchar(50);comment:'学历'" json:"education"` // 学历  
 WorkUnit string `gorm:"type:varchar(100);comment:'工作单位'" json:"workUnit"` // 工作单位  
 Title string `gorm:"type:varchar(50);comment:'职称'" json:"title"` // 职称  
 Experience string `gorm:"type:text;comment:'从艺经历'" json:"experience"` // 从艺经历  
 Honor string `gorm:"type:text;comment:'获得荣誉'" json:"honor"` // 获得荣誉  
 Speciality string `gorm:"type:text;comment:'特长技艺'" json:"speciality"` // 特长技艺  
 Representative bool `gorm:"type:tinyint(1);comment:'是否代表性传承人'" json:"representative"` // 是否代表性传承人  
}

package model  
  
import "gorm.io/gorm"  
  
// PasswordHistory 密码历史记录表  
type PasswordHistory struct {  
 gorm.Model  
 UserID uint `gorm:"type:int;comment:'用户ID'" json:"userId"` // 用户ID  
 Username string `gorm:"type:varchar(50);comment:'用户名'" json:"username"` // 用户名  
 Password string `gorm:"type:varchar(255);comment:'密码'" json:"-"` // 密码(加密存储)  
 ChangeTime string `gorm:"type:varchar(20);comment:'修改时间'" json:"changeTime"` // 修改时间  
 ExpireTime string `gorm:"type:varchar(20);comment:'过期时间'" json:"expireTime"` // 过期时间  
 ChangedBy string `gorm:"type:varchar(50);comment:'修改人'" json:"changedBy"` // 修改人  
 ChangeType string `gorm:"type:varchar(20);comment:'修改类型'" json:"changeType"` // 修改类型(重置/修改)  
 Description string `gorm:"type:varchar(255);comment:'说明'" json:"description"` // 说明  
}

package model  
  
import (  
 "gorm.io/gorm"  
)  
  
type Menu struct {  
 gorm.Model  
 Name string `gorm:"type:varchar(50);comment:'菜单名称(英文名, 可用于国际化)'" json:"name"`  
 Title string `gorm:"type:varchar(50);comment:'菜单标题(无法国际化时使用)'" json:"title"`  
 Icon string `gorm:"type:varchar(50);comment:'菜单图标'" json:"icon"`  
 Path string `gorm:"type:varchar(100);comment:'菜单访问路径'" json:"path"`  
 Redirect string `gorm:"type:varchar(100);comment:'重定向路径'" json:"redirect"`  
 Component string `gorm:"type:varchar(100);comment:'前端组件路径'" json:"component"`  
 Sort uint `gorm:"type:int(3);default:999;comment:'菜单顺序(1-999)'" json:"sort"`  
 Status uint `gorm:"type:tinyint(1);default:1;comment:'菜单状态(正常/禁用, 默认正常)'" json:"status"`  
 Hidden uint `gorm:"type:tinyint(1);default:2;comment:'菜单在侧边栏隐藏(1隐藏，2显示)'" json:"hidden"`  
 NoCache uint `gorm:"type:tinyint(1);default:2;comment:'菜单是否被 <keep-alive> 缓存(1不缓存，2缓存)'" json:"noCache"`  
 AlwaysShow uint `gorm:"type:tinyint(1);default:2;comment:'忽略之前定义的规则，一直显示根路由(1忽略，2不忽略)'" json:"alwaysShow"`  
 Breadcrumb uint `gorm:"type:tinyint(1);default:1;comment:'面包屑可见性(可见/隐藏, 默认可见)'" json:"breadcrumb"`  
 ActiveMenu string `gorm:"type:varchar(100);comment:'在其它路由时，想在侧边栏高亮的路由'" json:"activeMenu"`  
 ParentId uint `gorm:"default:0;comment:'父菜单编号(编号为0时表示根菜单)'" json:"parentId"`  
 Creator string `gorm:"type:varchar(20);comment:'创建人'" json:"creator"`  
 Children []\*Menu `gorm:"-" json:"children"` // 子菜单集合  
 Roles []\*Role `gorm:"many2many:role\_menus;" json:"roles"` // 角色菜单多对多关系  
}

package model  
  
import "gorm.io/gorm"  
  
type Api struct {  
 gorm.Model  
 Method string `gorm:"type:varchar(20);comment:'请求方式'" json:"method"`  
 Path string `gorm:"type:varchar(100);comment:'访问路径'" json:"path"`  
 Category string `gorm:"type:varchar(50);comment:'所属类别'" json:"category"`  
 Remark string `gorm:"type:varchar(100);comment:'备注'" json:"remark"`  
 Creator string `gorm:"type:varchar(20);comment:'创建人'" json:"creator"`  
}

package model  
  
import "gorm.io/gorm"  
  
// LoginHistory 登录历史记录表  
type LoginHistory struct {  
 gorm.Model  
 UserID uint `gorm:"type:int;comment:'用户ID'" json:"userId"` // 用户ID  
 Username string `gorm:"type:varchar(50);comment:'用户名'" json:"username"` // 用户名  
 LoginTime string `gorm:"type:varchar(20);comment:'登录时间'" json:"loginTime"` // 登录时间  
 LogoutTime string `gorm:"type:varchar(20);comment:'登出时间'" json:"logoutTime"` // 登出时间  
 IP string `gorm:"type:varchar(50);comment:'IP地址'" json:"ip"` // IP地址  
 Location string `gorm:"type:varchar(100);comment:'登录地点'" json:"location"` // 登录地点  
 Device string `gorm:"type:varchar(50);comment:'登录设备'" json:"device"` // 登录设备  
 Browser string `gorm:"type:varchar(50);comment:'浏览器'" json:"browser"` // 浏览器  
 Status int `gorm:"type:int;comment:'登录状态'" json:"status"` // 登录状态(0:失败,1:成功)  
 ErrorMessage string `gorm:"type:varchar(255);comment:'错误信息'" json:"errorMessage"` // 错误信息  
}

package model  
  
import (  
 "gorm.io/datatypes"  
 "gorm.io/gorm"  
)  
  
type FieldRelation struct {  
 gorm.Model  
 Flag string  
 Attributes datatypes.JSON  
}

package response  
  
import "github.com/eryajf/go-ldap-admin/model"  
  
type RoleListRsp struct {  
 Total int64 `json:"total"`  
 Roles []model.Role `json:"roles"`  
}

package response  
  
import "github.com/eryajf/go-ldap-admin/model"  
  
type ApiTreeRsp struct {  
 ID int `json:"ID"`  
 Remark string `json:"remark"`  
 Category string `json:"category"`  
 Children []\*model.Api `json:"children"`  
}  
  
type ApiListRsp struct {  
 Total int64 `json:"total"`  
 Apis []model.Api `json:"apis"`  
}

package response  
  
import "github.com/eryajf/go-ldap-admin/model"  
  
type LogListRsp struct {  
 Total int64 `json:"total"`  
 Logs []model.OperationLog `json:"logs"`  
}

package response  
  
import (  
 "net/http"  
  
 "github.com/gin-gonic/gin"  
)  
  
// 返回前端  
func Response(c \*gin.Context, httpStatus int, code int, data gin.H, message string) {  
 c.JSON(httpStatus, gin.H{  
 "code": code,  
 "data": data,  
 "message": message,  
 })  
}  
  
// 返回前端-成功  
func Success(c \*gin.Context, data gin.H, message string) {  
 Response(c, http.StatusOK, 200, data, message)  
}  
  
// 返回前端-失败  
func Fail(c \*gin.Context, data gin.H, message string) {  
 Response(c, http.StatusBadRequest, 400, data, message)  
}

package response  
  
import "github.com/eryajf/go-ldap-admin/model"  
  
type UserListRsp struct {  
 Total int `json:"total"`  
 Users []model.User `json:"users"`  
}

package response  
  
import "github.com/eryajf/go-ldap-admin/model"  
  
type MenuListRsp struct {  
 Total int64 `json:"total"`  
 Menus []model.Menu `json:"menus"`  
}

package response  
  
type DashboardList struct {  
 DataType string `json:"dataType"`  
 DataName string `json:"dataName"`  
 DataCount int64 `json:"dataCount"`  
 Icon string `json:"icon"`  
 Path string `json:"path"`  
}

package response  
  
import "github.com/eryajf/go-ldap-admin/model"  
  
type GroupListRsp struct {  
 Total int64 `json:"total"`  
 Groups []model.Group `json:"groups"`  
}  
  
type Guser struct {  
 UserId int64 `json:"userId"`  
 UserName string `json:"userName"`  
 NickName string `json:"nickName"`  
 Mail string `json:"mail"`  
 JobNumber string `json:"jobNumber"`  
 Mobile string `json:"mobile"`  
 Introduction string `json:"introduction"`  
}  
  
type GroupUsers struct {  
 GroupId int64 `json:"groupId"`  
 GroupName string `json:"groupName"`  
 GroupRemark string `json:"groupRemark"`  
 UserList []Guser `json:"userList"`  
}

package response  
  
/\*\*  
 \* @Author: 南宫乘风  
 \* @Description:  
 \* @File: responsebody.go  
 \* @Email: 1794748404@qq.com  
 \* @Date: 2024-05-17 16:24  
 \*/  
  
type ResponseBody struct {  
 Code int `json:"code"`  
 Msg string `json:"msg"`  
 Data interface{} `json:"data"`  
}

package request  
  
// PasswordPolicyReq 密码策略请求结构体  
type PasswordPolicyReq struct {  
 MinLength int `json:"minLength" validate:"required,min=6,max=32"` // 最小长度  
 RequireUppercase bool `json:"requireUppercase"` // 要求大写字母  
 RequireLowercase bool `json:"requireLowercase"` // 要求小写字母  
 RequireNumber bool `json:"requireNumber"` // 要求数字  
 RequireSpecialChar bool `json:"requireSpecialChar"` // 要求特殊字符  
 MaxAge int `json:"maxAge" validate:"required,min=0"` // 密码最大有效期(天)  
 HistoryCount int `json:"historyCount" validate:"required,min=0,max=24"` // 密码历史记录数  
 LockoutThreshold int `json:"lockoutThreshold" validate:"required,min=0,max=10"` // 锁定阈值  
 LockoutDuration int `json:"lockoutDuration" validate:"required,min=0,max=1440"` // 锁定时间(分钟)  
}

package request  
  
// FieldRelationListReq 获取资源列表结构体  
type FieldRelationListReq struct {  
}  
  
// FieldRelationAddReq 添加资源结构体  
type FieldRelationAddReq struct {  
 Flag string `json:"flag" validate:"required,min=1,max=20"`  
 Attributes map[string]string `json:"attributes" validate:"required,gt=0"`  
}  
  
// FieldRelationUpdateReq 更新资源结构体  
type FieldRelationUpdateReq struct {  
 ID uint `json:"id" validate:"required"`  
 Flag string `json:"flag" validate:"required,min=1,max=20"`  
 Attributes map[string]string `json:"attributes" validate:"required,gt=0"`  
}  
  
// FieldRelationDeleteReq 删除资源结构体  
type FieldRelationDeleteReq struct {  
 FieldRelationIds []uint `json:"fieldRelationIds" validate:"required"`  
}

package request  
  
// ApiListReq 获取资源列表结构体  
type ApiListReq struct {  
 Method string `json:"method" form:"method"`  
 Path string `json:"path" form:"path"`  
 Category string `json:"category" form:"category"`  
 Creator string `json:"creator" form:"creator"`  
 PageNum int `json:"pageNum" form:"pageNum"`  
 PageSize int `json:"pageSize" form:"pageSize"`  
}  
  
// ApiAddReq 添加资源结构体  
type ApiAddReq struct {  
 Method string `json:"method" validate:"required,min=1,max=20"`  
 Path string `json:"path" validate:"required,min=1,max=100"`  
 Category string `json:"category" validate:"required,min=1,max=50"`  
 Remark string `json:"remark" validate:"min=0,max=100"`  
}  
  
// ApiUpdateReq 更新资源结构体  
type ApiUpdateReq struct {  
 ID uint `json:"id" validate:"required"`  
 Method string `json:"method" validate:"min=1,max=20"`  
 Path string `json:"path" validate:"min=1,max=100"`  
 Category string `json:"category" validate:"min=1,max=50"`  
 Remark string `json:"remark" validate:"min=0,max=100"`  
}  
  
// ApiDeleteReq 删除资源结构体  
type ApiDeleteReq struct {  
 ApiIds []uint `json:"apiIds" validate:"required"`  
}  
  
// ApiGetTreeReq 获取资源树结构体  
type ApiGetTreeReq struct {  
}

package request  
  
// OperationLogListReq 操作日志请求结构体  
type OperationLogListReq struct {  
 Username string `json:"username" form:"username"`  
 Ip string `json:"ip" form:"ip"`  
 Path string `json:"path" form:"path"`  
 Method string `json:"method" form:"method"`  
 Status int `json:"status" form:"status"`  
 PageNum int `json:"pageNum" form:"pageNum"`  
 PageSize int `json:"pageSize" form:"pageSize"`  
}  
  
// OperationLogDeleteReq 批量删除操作日志结构体  
type OperationLogDeleteReq struct {  
 OperationLogIds []uint `json:"operationLogIds" validate:"required"`  
}

package request  
  
// RoleAddReq 添加资源结构体  
type RoleAddReq struct {  
 Name string `json:"name" validate:"required,min=1,max=20"`  
 Keyword string `json:"keyword" validate:"required,min=1,max=20"`  
 Remark string `json:"remark" validate:"min=0,max=100"`  
 Status uint `json:"status" validate:"oneof=1 2"`  
 Sort uint `json:"sort" validate:"gte=1,lte=999"`  
}  
  
// RoleListReq 列表结构体  
type RoleListReq struct {  
 Name string `json:"name" form:"name"`  
 Keyword string `json:"keyword" form:"keyword"`  
 Status uint `json:"status" form:"status"`  
 PageNum int `json:"pageNum" form:"pageNum"`  
 PageSize int `json:"pageSize" form:"pageSize"`  
}  
  
// RoleUpdateReq 更新资源结构体  
type RoleUpdateReq struct {  
 ID uint `json:"id" validate:"required"`  
 Name string `json:"name" validate:"required,min=1,max=20"`  
 Keyword string `json:"keyword" validate:"required,min=1,max=20"`  
 Remark string `json:"remark" validate:"min=0,max=100"`  
 Status uint `json:"status" validate:"oneof=1 2"`  
 Sort uint `json:"sort" validate:"gte=1,lte=999"`  
}  
  
// RoleDeleteReq 删除资源结构体  
type RoleDeleteReq struct {  
 RoleIds []uint `json:"roleIds" validate:"required"`  
}  
  
// RoleGetTreeReq 获取资源树结构体  
type RoleGetTreeReq struct {  
}  
  
// RoleGetMenuListReq 获取角色菜单列表结构体  
type RoleGetMenuListReq struct {  
 RoleID uint `json:"roleId" form:"roleId" validate:"required"`  
}  
  
// RoleGetApiListReq 获取角色接口列表结构体  
type RoleGetApiListReq struct {  
 RoleID uint `json:"roleId" form:"roleId" validate:"required"`  
}  
  
// RoleUpdateMenusReq 更新角色菜单结构体  
type RoleUpdateMenusReq struct {  
 RoleID uint `json:"roleId" validate:"required"`  
 MenuIds []uint `json:"menuIds" validate:"required"`  
}  
  
// RoleUpdateApisReq 更新角色接口结构体  
type RoleUpdateApisReq struct {  
 RoleID uint `json:"roleId" validate:"required"`  
 ApiIds []uint `json:"apiIds" validate:"required"`  
}

package request  
  
// LDAPMonitorReq LDAP监控请求结构体  
type LDAPMonitorReq struct {  
 ConnectionStatus bool `json:"connectionStatus"` // 连接状态  
 ResponseTime int64 `json:"responseTime"` // 响应时间(ms)  
 LastCheckTime string `json:"lastCheckTime"` // 最后检查时间  
 ErrorCount int `json:"errorCount"` // 错误次数  
 LastError string `json:"lastError"` // 最后错误信息  
}

package request  
  
// BaseSendCodeReq 发送验证码  
type BaseSendCodeReq struct {  
 Mail string `json:"mail" validate:"required,min=0,max=100"`  
}  
  
// BaseChangePwdReq 修改密码结构体  
type BaseChangePwdReq struct {  
 Mail string `json:"mail" validate:"required,min=0,max=100"`  
 Code string `json:"code" validate:"required,len=6"`  
}  
  
// BaseDashboardReq 系统首页展示数据结构体  
type BaseDashboardReq struct {  
}  
  
// EncryptPasswdReq  
type EncryptPasswdReq struct {  
 Passwd string `json:"passwd" form:"passwd" validate:"required"`  
}  
  
// DecryptPasswdReq  
type DecryptPasswdReq struct {  
 Passwd string `json:"passwd" form:"passwd" validate:"required"`  
}

package request  
  
// MenuAddReq 添加资源结构体  
type MenuAddReq struct {  
 Name string `json:"name" validate:"required,min=1,max=50"`  
 Title string `json:"title" validate:"required,min=1,max=50"`  
 Icon string `json:"icon" validate:"min=0,max=50"`  
 Path string `json:"path" validate:"required,min=1,max=100"`  
 Redirect string `json:"redirect" validate:"min=0,max=100"`  
 Component string `json:"component" validate:"required,min=1,max=100"`  
 Sort uint `json:"sort" validate:"gte=1,lte=999"`  
 Status uint `json:"status" validate:"oneof=1 2"`  
 Hidden uint `json:"hidden" validate:"oneof=1 2"`  
 NoCache uint `json:"noCache" validate:"oneof=1 2"`  
 AlwaysShow uint `json:"alwaysShow" validate:"oneof=1 2"`  
 Breadcrumb uint `json:"breadcrumb" validate:"oneof=1 2"`  
 ActiveMenu string `json:"activeMenu" validate:"min=0,max=100"`  
 ParentId uint `json:"parentId"`  
}  
  
// MenuListReq 列表结构体  
type MenuListReq struct {  
}  
  
// MenuUpdateReq 更新资源结构体  
type MenuUpdateReq struct {  
 ID uint `json:"id" validate:"required"`  
 Name string `json:"name" validate:"required,min=1,max=50"`  
 Title string `json:"title" validate:"required,min=1,max=50"`  
 Icon string `json:"icon" validate:"min=0,max=50"`  
 Path string `json:"path" validate:"required,min=1,max=100"`  
 Redirect string `json:"redirect" validate:"min=0,max=100"`  
 Component string `json:"component" validate:"min=0,max=100"`  
 Sort uint `json:"sort" validate:"gte=1,lte=999"`  
 Status uint `json:"status" validate:"oneof=1 2"`  
 Hidden uint `json:"hidden" validate:"oneof=1 2"`  
 NoCache uint `json:"noCache" validate:"oneof=1 2"`  
 AlwaysShow uint `json:"alwaysShow" validate:"oneof=1 2"`  
 Breadcrumb uint `json:"breadcrumb" validate:"oneof=1 2"`  
 ActiveMenu string `json:"activeMenu" validate:"min=0,max=100"`  
 ParentId uint `json:"parentId" validate:"gte=0"`  
}  
  
// MenuDeleteReq 删除资源结构体  
type MenuDeleteReq struct {  
 MenuIds []uint `json:"menuIds" validate:"required"`  
}  
  
// MenuGetTreeReq 获取菜单树结构体  
type MenuGetTreeReq struct {  
}  
  
// MenuGetAccessTreeReq 获取用户菜单树  
type MenuGetAccessTreeReq struct {  
 ID uint `json:"id" form:"id"`  
}

package request  
  
// UserAddReq 创建资源结构体  
type UserAddReq struct {  
 Username string `json:"username" validate:"required,min=2,max=50"`  
 Password string `json:"password"`  
 Nickname string `json:"nickname" validate:"required,min=0,max=50"`  
 GivenName string `json:"givenName" validate:"min=0,max=50"`  
 Mail string `json:"mail" validate:"required,min=0,max=100"`  
 JobNumber string `json:"jobNumber" validate:"required,min=0,max=20"`  
 PostalAddress string `json:"postalAddress" validate:"min=0,max=255"`  
 Departments string `json:"departments" validate:"min=0,max=512"`  
 Position string `json:"position" validate:"min=0,max=128"`  
 Mobile string `json:"mobile" validate:"required,checkMobile"`  
 Avatar string `json:"avatar"`  
 Introduction string `json:"introduction" validate:"min=0,max=255"`  
 Status uint `json:"status" validate:"oneof=1 2"`  
 DepartmentId []uint `json:"departmentId" validate:"required"`  
 Source string `json:"source" validate:"min=0,max=50"`  
 RoleIds []uint `json:"roleIds" validate:"required"`  
}  
  
// DingUserAddReq 钉钉用户创建资源结构体  
type DingUserAddReq struct {  
 Username string `json:"username" validate:"required,min=2,max=50"`  
 Password string `json:"password"`  
 Nickname string `json:"nickname" validate:"required,min=0,max=50"`  
 GivenName string `json:"givenName" validate:"min=0,max=50"`  
 Mail string `json:"mail" validate:"required,min=0,max=100"`  
 JobNumber string `json:"jobNumber" validate:"required,min=0,max=20"`  
 PostalAddress string `json:"postalAddress" validate:"min=0,max=255"`  
 Departments string `json:"departments" validate:"min=0,max=512"`  
 Position string `json:"position" validate:"min=0,max=128"`  
 Mobile string `json:"mobile" validate:"required,checkMobile"`  
 Avatar string `json:"avatar"`  
 Introduction string `json:"introduction" validate:"min=0,max=255"`  
 Status uint `json:"status" validate:"oneof=1 2"`  
 DepartmentId []uint `json:"departmentId" validate:"required"`  
 Source string `json:"source" validate:"min=0,max=50"`  
 RoleIds []uint `json:"roleIds" validate:"required"`  
 SourceUserId string `json:"sourceUserId"` // 第三方用户id  
 SourceUnionId string `json:"sourceUnionId"` // 第三方唯一unionId  
}  
  
// WeComUserAddReq 企业微信用户创建资源结构体  
type WeComUserAddReq struct {  
 Username string `json:"username" validate:"required,min=2,max=50"`  
 Password string `json:"password"`  
 Nickname string `json:"nickname" validate:"required,min=0,max=50"`  
 GivenName string `json:"givenName" validate:"min=0,max=50"`  
 Mail string `json:"mail" validate:"required,min=0,max=100"`  
 JobNumber string `json:"jobNumber" validate:"required,min=0,max=20"`  
 PostalAddress string `json:"postalAddress" validate:"min=0,max=255"`  
 Departments string `json:"departments" validate:"min=0,max=512"`  
 Position string `json:"position" validate:"min=0,max=128"`  
 Mobile string `json:"mobile" validate:"required,checkMobile"`  
 Avatar string `json:"avatar"`  
 Introduction string `json:"introduction" validate:"min=0,max=255"`  
 Status uint `json:"status" validate:"oneof=1 2"`  
 DepartmentId []uint `json:"departmentId" validate:"required"`  
 Source string `json:"source" validate:"min=0,max=50"`  
 RoleIds []uint `json:"roleIds" validate:"required"`  
 SourceUserId string `json:"sourceUserId"` // 第三方用户id  
 SourceUnionId string `json:"sourceUnionId"` // 第三方唯一unionId  
}  
  
// UserUpdateReq 更新资源结构体  
type UserUpdateReq struct {  
 ID uint `json:"id" validate:"required"`  
 Username string `json:"username" validate:"required,min=2,max=50"`  
 Nickname string `json:"nickname" validate:"min=0,max=20"`  
 GivenName string `json:"givenName" validate:"min=0,max=50"`  
 Mail string `json:"mail" validate:"min=0,max=100"`  
 JobNumber string `json:"jobNumber" validate:"min=0,max=20"`  
 PostalAddress string `json:"postalAddress" validate:"min=0,max=255"`  
 Departments string `json:"departments" validate:"min=0,max=512"`  
 Position string `json:"position" validate:"min=0,max=128"`  
 Mobile string `json:"mobile" validate:"checkMobile"`  
 Avatar string `json:"avatar"`  
 Introduction string `json:"introduction" validate:"min=0,max=255"`  
 DepartmentId []uint `json:"departmentId" validate:"required"`  
 Source string `json:"source" validate:"min=0,max=50"`  
 RoleIds []uint `json:"roleIds" validate:"required"`  
}  
  
// UserDeleteReq 批量删除资源结构体  
type UserDeleteReq struct {  
 UserIds []uint `json:"userIds" validate:"required"`  
}  
  
// UserChangePwdReq 修改密码结构体  
type UserChangePwdReq struct {  
 OldPassword string `json:"oldPassword" validate:"required"`  
 NewPassword string `json:"newPassword" validate:"required"`  
}  
  
// UserChangeUserStatusReq 修改用户状态结构体  
type UserChangeUserStatusReq struct {  
 ID uint `json:"id" validate:"required"`  
 Status uint `json:"status" validate:"oneof=1 2"`  
}  
  
// UserGetUserInfoReq 获取用户信息结构体  
type UserGetUserInfoReq struct {  
}  
  
// SyncDingUserReq 同步钉钉用户信息  
type SyncDingUserReq struct {  
}  
  
// SyncWeComUserReq 同步企业微信用户信息  
type SyncWeComUserReq struct {  
}  
  
// SyncFeiShuUserReq 同步飞书用户信息  
type SyncFeiShuUserReq struct {  
}  
  
// SyncOpenLdapUserReq 同步ldap用户信息  
type SyncOpenLdapUserReq struct {  
}  
type SyncSqlUserReq struct {  
 UserIds []uint `json:"userIds" validate:"required"`  
}  
  
// UserListReq 获取用户列表结构体  
type UserListReq struct {  
 Username string `json:"username" form:"username"`  
 Mobile string `json:"mobile" form:"mobile" `  
 Nickname string `json:"nickname" form:"nickname"`  
 GivenName string `json:"givenName" form:"givenName"`  
 DepartmentId []uint `json:"departmentId" form:"departmentId"`  
 Status uint `json:"status" form:"status" `  
 SyncState uint `json:"syncState" form:"syncState" `  
 PageNum int `json:"pageNum" form:"pageNum"`  
 PageSize int `json:"pageSize" form:"pageSize"`  
}  
  
// RegisterAndLoginReq 用户登录结构体  
type RegisterAndLoginReq struct {  
 Username string `form:"username" json:"username" binding:"required"`  
 Password string `form:"password" json:"password" binding:"required"`  
}

package request  
  
// UserImportReq 用户导入请求结构体  
type UserImportReq struct {  
 Users []UserImportItem `json:"users" validate:"required,dive"` // 用户列表  
}  
  
// UserImportItem 用户导入项  
type UserImportItem struct {  
 Username string `json:"username" validate:"required"` // 用户名  
 Nickname string `json:"nickname" validate:"required"` // 昵称  
 Email string `json:"email" validate:"required,email"` // 邮箱  
 Phone string `json:"phone" validate:"required"` // 手机号  
 Department string `json:"department"` // 部门  
 Position string `json:"position"` // 职位  
 Description string `json:"description"` // 描述  
}

package request  
  
// GroupListReq 获取资源列表结构体  
type GroupListReq struct {  
 GroupName string `json:"groupName" form:"groupName"`  
 Remark string `json:"remark" form:"remark"`  
 PageNum int `json:"pageNum" form:"pageNum"`  
 PageSize int `json:"pageSize" form:"pageSize"`  
 SyncState uint `json:"syncState" form:"syncState"`  
}  
  
// GroupListAllReq 获取资源列表结构体，不分页  
type GroupListAllReq struct {  
 GroupName string `json:"groupName" form:"groupName"`  
 GroupType string `json:"groupType" form:"groupType"`  
 Remark string `json:"remark" form:"remark"`  
 Source string `json:"source" form:"source"`  
 SourceDeptId string `json:"sourceDeptId"`  
 SourceDeptParentId string `json:"SourceDeptParentId"`  
}  
  
// GroupAddReq 添加资源结构体  
type GroupAddReq struct {  
 GroupType string `json:"groupType" validate:"required,min=1,max=20"`  
 GroupName string `json:"groupName" validate:"required,min=1,max=128"`  
 //父级Id 大于等于0 必填  
 ParentId uint `json:"parentId" validate:"omitempty,min=0"`  
 Remark string `json:"remark" validate:"min=0,max=128"` // 分组的中文描述  
}  
  
// DingTalkGroupAddReq 添加钉钉资源结构体  
type DingGroupAddReq struct {  
 GroupType string `json:"groupType" validate:"required,min=1,max=20"`  
 GroupName string `json:"groupName" validate:"required,min=1,max=128"`  
 //父级Id 大于等于0 必填  
 ParentId uint `json:"parentId" validate:"omitempty,min=0"`  
 Remark string `json:"remark" validate:"min=0,max=128"` // 分组的中文描述  
 SourceDeptId string `json:"sourceDeptId"`  
 Source string `json:"source"`  
 SourceDeptParentId string `json:"SourceDeptParentId"`  
 SourceUserNum int `json:"sourceUserNum"`  
}  
  
// WeComGroupAddReq 添加企业微信资源结构体  
type WeComGroupAddReq struct {  
 GroupType string `json:"groupType" validate:"required,min=1,max=20"`  
 GroupName string `json:"groupName" validate:"required,min=1,max=128"`  
 //父级Id 大于等于0 必填  
 ParentId uint `json:"parentId" validate:"omitempty,min=0"`  
 Remark string `json:"remark" validate:"min=0,max=128"` // 分组的中文描述  
 SourceDeptId string `json:"sourceDeptId"`  
 Source string `json:"source"`  
 SourceDeptParentId string `json:"SourceDeptParentId"`  
 SourceUserNum int `json:"sourceUserNum"`  
}  
  
// GroupUpdateReq 更新资源结构体  
type GroupUpdateReq struct {  
 ID uint `json:"id" form:"id" validate:"required"`  
 GroupName string `json:"groupName" validate:"required,min=1,max=128"`  
 Remark string `json:"remark" validate:"min=0,max=128"` // 分组的中文描述  
}  
  
// GroupDeleteReq 删除资源结构体  
type GroupDeleteReq struct {  
 GroupIds []uint `json:"groupIds" validate:"required"`  
}  
  
// GroupGetTreeReq 获取资源树结构体  
type GroupGetTreeReq struct {  
 GroupName string `json:"groupName" form:"groupName"`  
 Remark string `json:"remark" form:"remark"`  
 PageNum int `json:"pageNum" form:"pageNum"`  
 PageSize int `json:"pageSize" form:"pageSize"`  
}  
  
type GroupAddUserReq struct {  
 GroupID uint `json:"groupId" validate:"required"`  
 UserIds []uint `json:"userIds" validate:"required"`  
}  
  
type GroupRemoveUserReq struct {  
 GroupID uint `json:"groupId" validate:"required"`  
 UserIds []uint `json:"userIds" validate:"required"`  
}  
  
// UserInGroupReq 在分组内的用户  
type UserInGroupReq struct {  
 GroupID uint `json:"groupId" form:"groupId" validate:"required"`  
 Nickname string `json:"nickname" form:"nickname"`  
}  
  
// UserNoInGroupReq 不在分组内的用户  
type UserNoInGroupReq struct {  
 GroupID uint `json:"groupId" form:"groupId" validate:"required"`  
 Nickname string `json:"nickname" form:"nickname"`  
}  
  
// SyncDingTalkDeptsReq 同步钉钉部门信息  
type SyncDingTalkDeptsReq struct {  
}  
  
// SyncWeComDeptsReq 同步企业微信部门信息  
type SyncWeComDeptsReq struct {  
}  
  
// SyncFeiShuDeptsReq 同步飞书部门信息  
type SyncFeiShuDeptsReq struct {  
}  
  
// SyncOpenLdapDeptsReq 同步原ldap部门信息  
type SyncOpenLdapDeptsReq struct {  
}  
  
// SyncOpenLdapDeptsReq 同步原ldap部门信息  
type SyncSqlGrooupsReq struct {  
 GroupIds []uint `json:"groupIds" validate:"required"`  
}