Cargo分析

requestContext

每调用一次Cargo，也就是调用Cargo的Invoke方法，都会创建一个新的requestContext

```
func NewContext(ctx context.Context, stub shim.ChaincodeStubInterface) Context {
    return &requestContext{
        context:        ctx,
        stub:           stub,
        currentChannel: stub.GetChannelID(),
    }
}
```

可以看到这个requestContext包括三个部分，一个是请求上下文，请求上下文当中可以放一些整个请求需要用到的key-value键值对，stub就是chaincode执行的stub，也是cargo调用其他chaincode需要的一个stub，currentChannel就是当前的channel，是一个字符串。

```
// cc router handler
background := context.WithValue(context.Background(), common.TraceID, val: "")
ctx := router.NewContext(background, stub)
```

```
var (
    background = new(emptyCtx)
    todo       = new(emptyCtx)
)

// Background returns a non-nil, empty Context. It is never canceled, has no
// values, and has no deadline. It is typically used by the main function,
// initialization, and tests, and as the top-level Context for incoming
// requests.
func Background() Context {
    return background
}
```

Context当中有一个很重要的方法，就是Value

```
    //  }
    Value(key interface{}) interface{}
}
```

这个方法可以从Context上下文对象当中读取上下文对象保存的key-value值。

创建了上下文对象之后，就可以把请求转发到调用其他的chaincode了，比如说cabbage、citrus等等：

```
func (r *router) dispatch(function string, ctx Context, args interface{}) result.Response {
    routeFunc, ok := r.routers[function]
    if ok {
        return routeFunc(ctx, &requestParam{
            function: function,
            param:    args,
        })
    }
    return result.Error(result.FUNCTION_NOT_EXIST)
}
```

话说这个routers里面究竟存储了什么呢，这就和chaincode和cargo的整合方式有关了，小明重写后的chaincode都有一个module目录，module里面都有一个init方法，在这个方法里面调用cargo实例的RegisterRouter方法来注册路由设置函数：

```
package module

import ...

var log = logging.GetLogger( pkg: "cabbage-router")

func init() {
    c := cargo.Get()

    c.InitFunc(func(param cargo.InitParam) error {...})

    c.RegisterRouter(read.Configure)
    c.RegisterRouter(write.Configure)
    c.RegisterRouter(node.Configure)
    c.RegisterRouter(config.Configure)
    c.RegisterRouter(white.Configure)
}
```

就拿这个read.Configure来说吧

```
func Configure(r router.Router) {
    h := new(handler)

    r.Register( function: "batchQueryData", h.batchQueryData)
    r.Register( function: "batchQueryBySupervise", h.batchQueryBySupervise)

    r.Register( function: "batchExistKey", h.batchExistKey)

    r.Register( function: "queryHistory", h.queryHistory)
    r.Register( function: "queryHistoryBySupervise", h.queryHistoryBySupervise)
    r.Register( function: "queryHistoryValueByKeyAndTxID", h.queryHistoryValueByKeyAndTxID)

    r.Register( function: "queryRangeKeys", h.queryRangeKeys)
    r.Register( function: "queryRangeData", h.queryRangeData)
    r.Register( function: "batchQueryRangeData", h.batchQueryRangeData)

    r.Register( function: "queryTxID", h.queryTxIds)
    r.Register( function: "queryLatestTxID", h.queryLatestTxID)
}
```

cargo的RegisterRouter方法会把chaincode的路由设置函数都放到一个routeGroup里面

```
func Get() *Cargo {
    cargoOnce.Do(func() {
        _cargo = &Cargo{
            f:          getFramework(),
            colorer:    color.New(),
            routeGroup: make([]router.RouteGroupFunc, 0),
        }
    })
    return _cargo
}
```

然后cargo启动的时候，会执行routeGroup里面所有的路由设置函数

```go
func (c *Cargo) Start(v common.Version) {

    c.colorer.Printf(banner, version, v)

    if len(c.routeGroup) == 0 {
        panic(fmt.Sprintf( format: "Start chaincode fail: not found router"))
    }

    c.f.Router.Use(c.routeGroup...)

    if e := shim.Start(c.f.cc); e != nil {
        panic(fmt.Sprintf( format: "Start chaincode fail: [%s]", e))
    }

}
```

执行路由注册函数

```go
func (r *router) Use(g ...RouteGroupFunc) {
    for _, f := range g {
        f(r)
    }
}
```

router的Register方法则会把方法名映射到具体的方法处理函数，这就是dispatch时候，根据请求参数中的方法名找到相应的处理函数的根据：

```go
func (r *router) Register(function string, routerFunc RouteFunc) {
    _, ok := r.routers[function]
    if ok {
        panic(fmt.Sprintf( format: "router duplication, function name:%s", function))
    } else {
        log.Infof( format: "function[%s] register successfully", function)
        r.routers[function] = routerFunc
    }
}
```