

- cc
 - device auth
 - device link
 - device operation
 - device rule
 - device-third-adapter
 - strategy-trigger
 - subscribe service
 - gossip

CC

项目结构介绍：服务一般包括五个模块，api模块是Controller rest API层，只定义了接口；core 模块是觉得的业务实现，包括api 层调用的业务组件实现和spi接口定义；model模块定义了服务的数据模型；client 模块定义了对外提供的api client;starter 模块引用api模块作为启动类，开启服务能力，同时实现了core模块的spi接口。

device auth

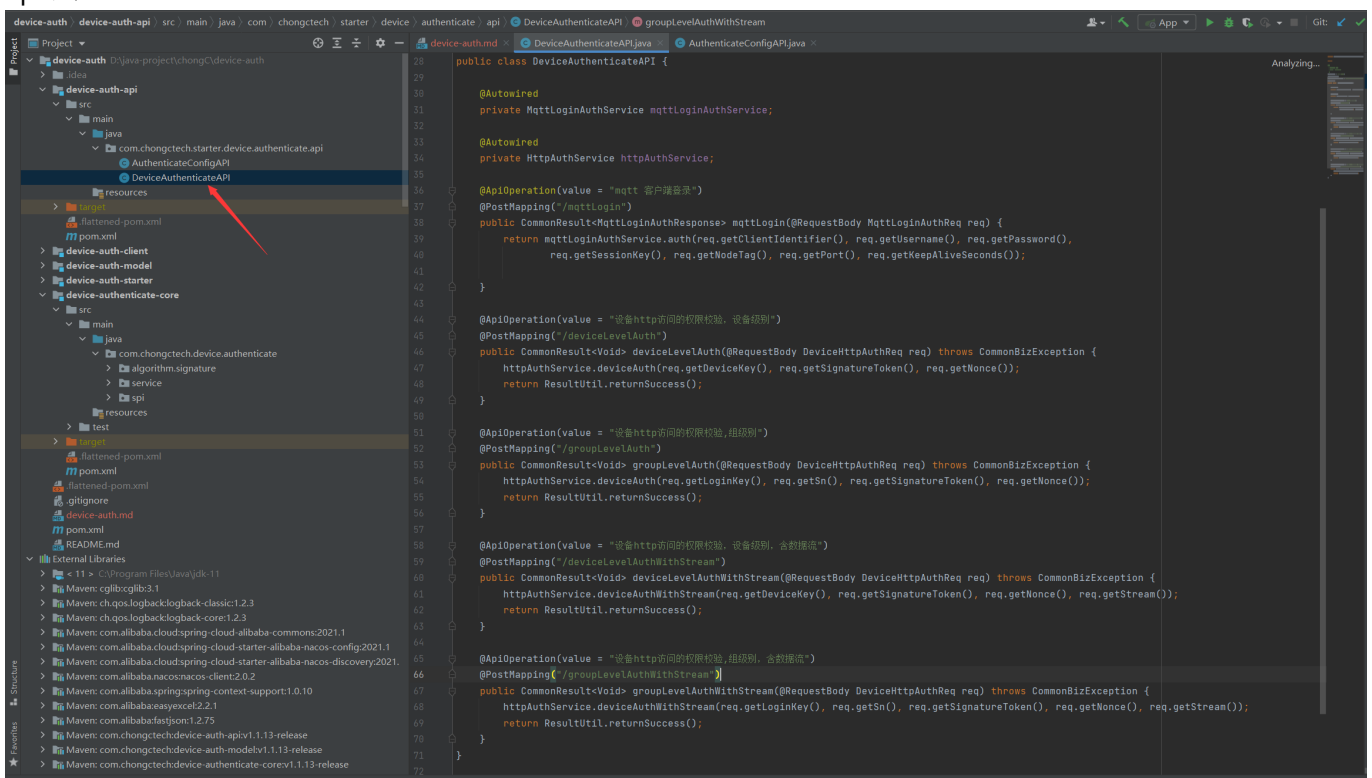
主要提供的是api服务：主要实现逻辑为平台鉴权信息校验，比如设备token,组token, app token 等。

DeviceAuthenticateAPI：mqtt broker 登录鉴权接口，分为设备鉴权，组鉴权，app鉴权，镜像鉴权;主要流程为先进行token校验，然后记录路由表，

然后生成welcome消息返回结果。网关http访问鉴权接口主要是是做token校验。

AuthenticateConfigAPI：mqtt 镜像登录鉴权配置接口，组级别鉴权配置接口，媒体播放令牌和鉴权接口；主要业务为设备信息校验，生成并记录配置信息返回。

api 入口



device link

包括两个模块：对外的rest api和mqtt 协议实现

1. DeviceLinkAPI： 提供了发送 qos 0和1的接口，踢设备下线接口和判断设备是否在线接口。主要是检测链路是否在线，然后构建报文下发。

2. Mqtt 协议实现

主要是通过接受mqtt消息进行报文解析，然后就是业务处理处理；

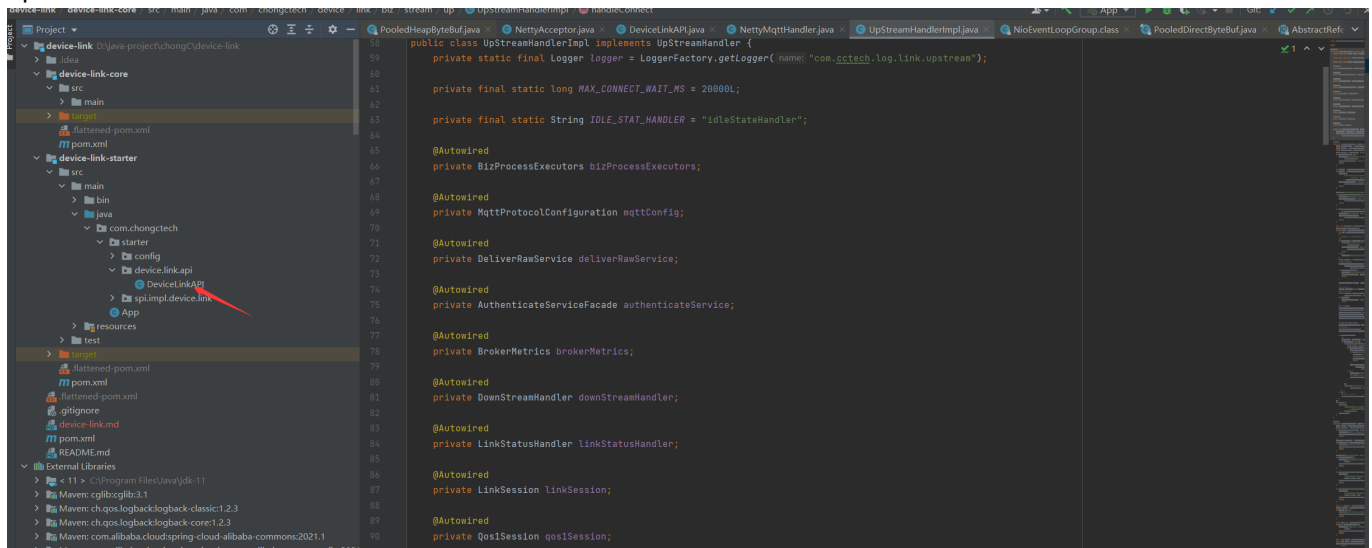
主要实现了Mqtt connect报文，对其做了身份认证和welcome消息发送；实现了mqtt publish qos 0 消息转发到pulsar, qos 1 报文的ack和pulsar转发；

实现了mqtt ping 消息回复和pulsar转发；实现了 mqtt disconnect 报文处理和puslar 转发；

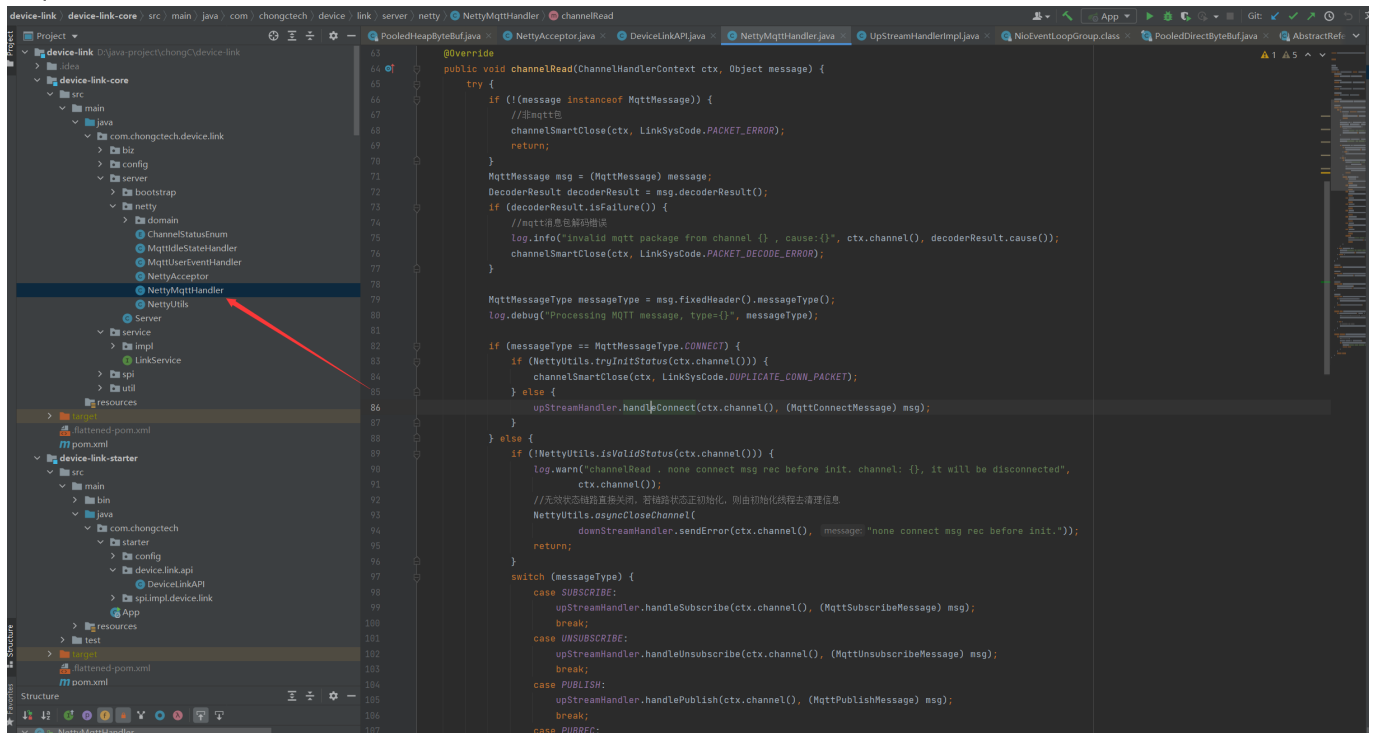
实现了平台下发mqtt publish报文到设备端,主要业务为mqtt报文发送和命令状态pulsar转发；

实现了平台idle 心跳check 事件和pulsar事件转发，主要通过链路ilde事件检测实现心跳事件的触发。

api 入口



mqtt 消息入口



device operation

该服务主要是提供device link 服务的代理，和sip 服务接口和命令生命周期处理。

CommandOperationAPI: 提供了命令下发接口，首先是查询路由表，然后存储数据，然后判断延迟命令，然后发送命令到设备，然后发送命令事件并实现域转发；

命令查询接口，分页查询接口和命令状态更新接口。

SipOperationAPI： 提供了sip服务进行invite和bye实现；其中并实现了srs 检测；

SysCommandOperationAPI： 提供了重置设备的命令接口；

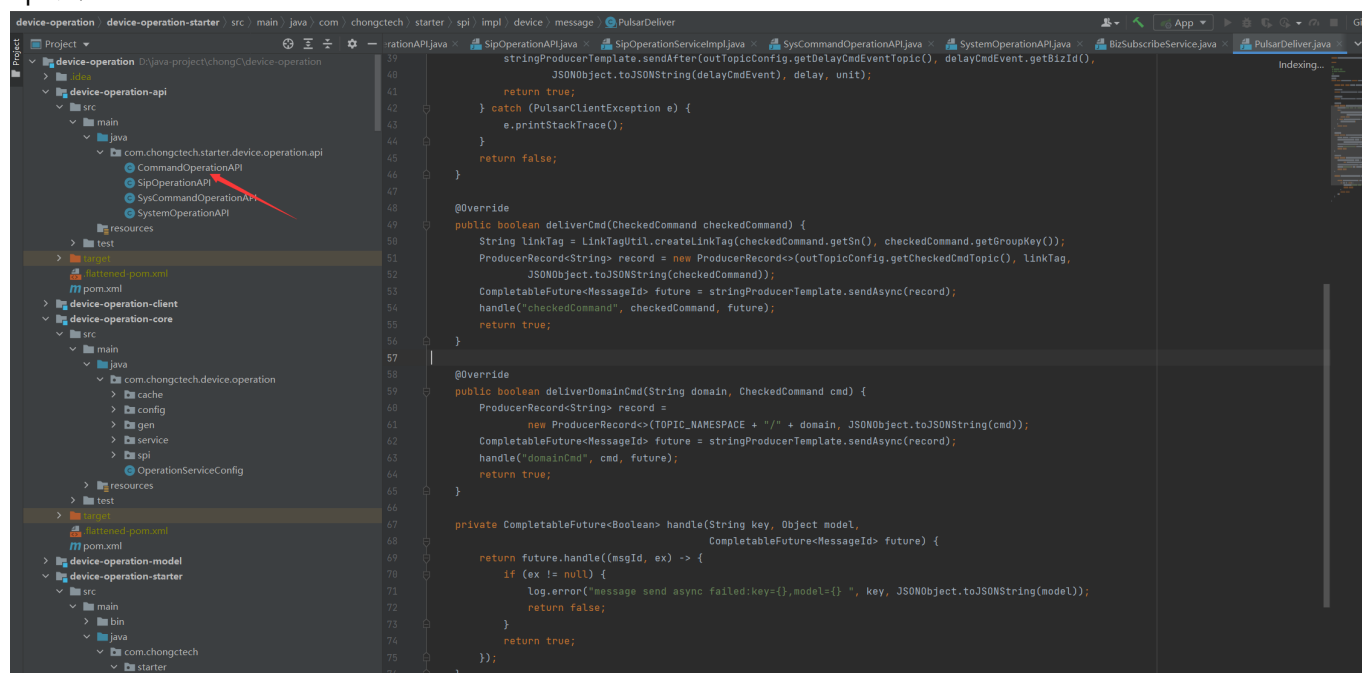
SystemOperationAPI： 系统命令中提供了上层服务对平台业务的通知，包括设备下线，订阅数据发送，

ota数据发送，设备模型发送，设备拓扑信息发送和影子状态发送。

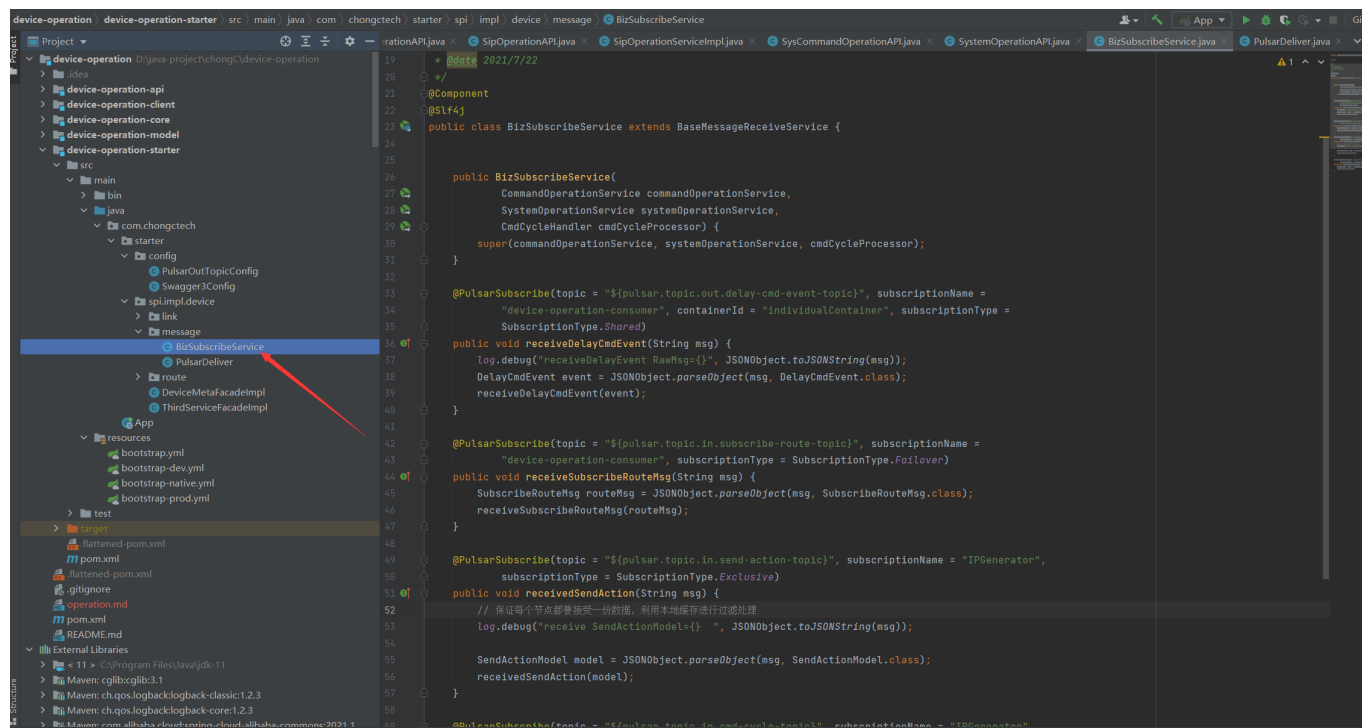
CmdCycleProcessorImpl： 命令的生命周期处理，主要是更新命令的状态，从低状态更新到高状态。

starter 模块为对外的接口实现和pulsar 消费数据处理

api 入口



事件入口



)

device rule

该服务主要提供了图片上传接口，平台mqtt 消息业务处理和路由实现。

1. `PictureAPI`：提供了图片上传接口

2. mqtt 消息业务处理：其中主要主要包括topic校验，数据加解密，代理校验，和topic业务处理。

包括了上行数据处理，命令处理和链路事件处理。上行数据处理：包括了设备档案同步处理，ota同步处理，设备影子查询处理，数据上报处理，批量数据上报处理，主题映射数据上报处理，设备模型拉取处理，设备拓扑关系处理；

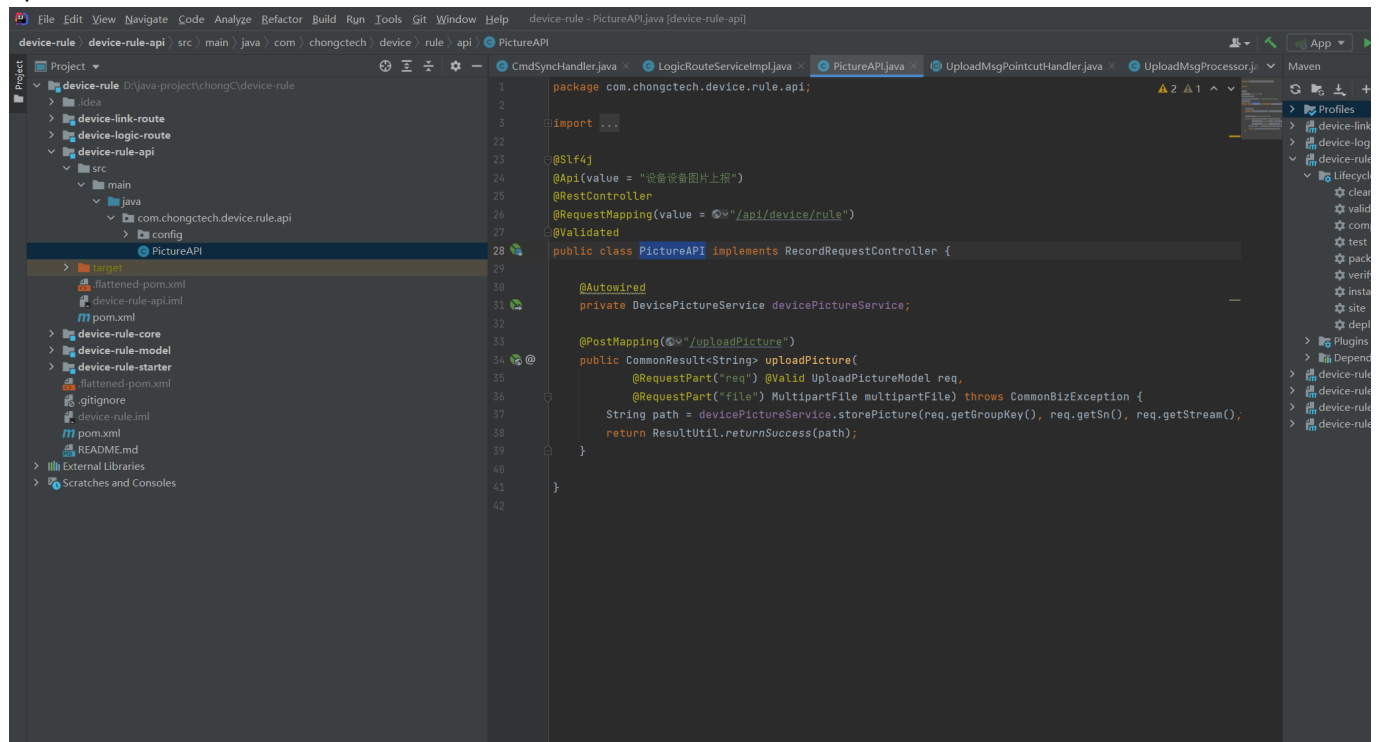
命令处理：设备侧上报命令执行结果事件pulsar发送。

链路处理：包括设备链路上线事件，下线事件，心跳事件，和平台心跳事件处理，主要是记录平台设备日志。

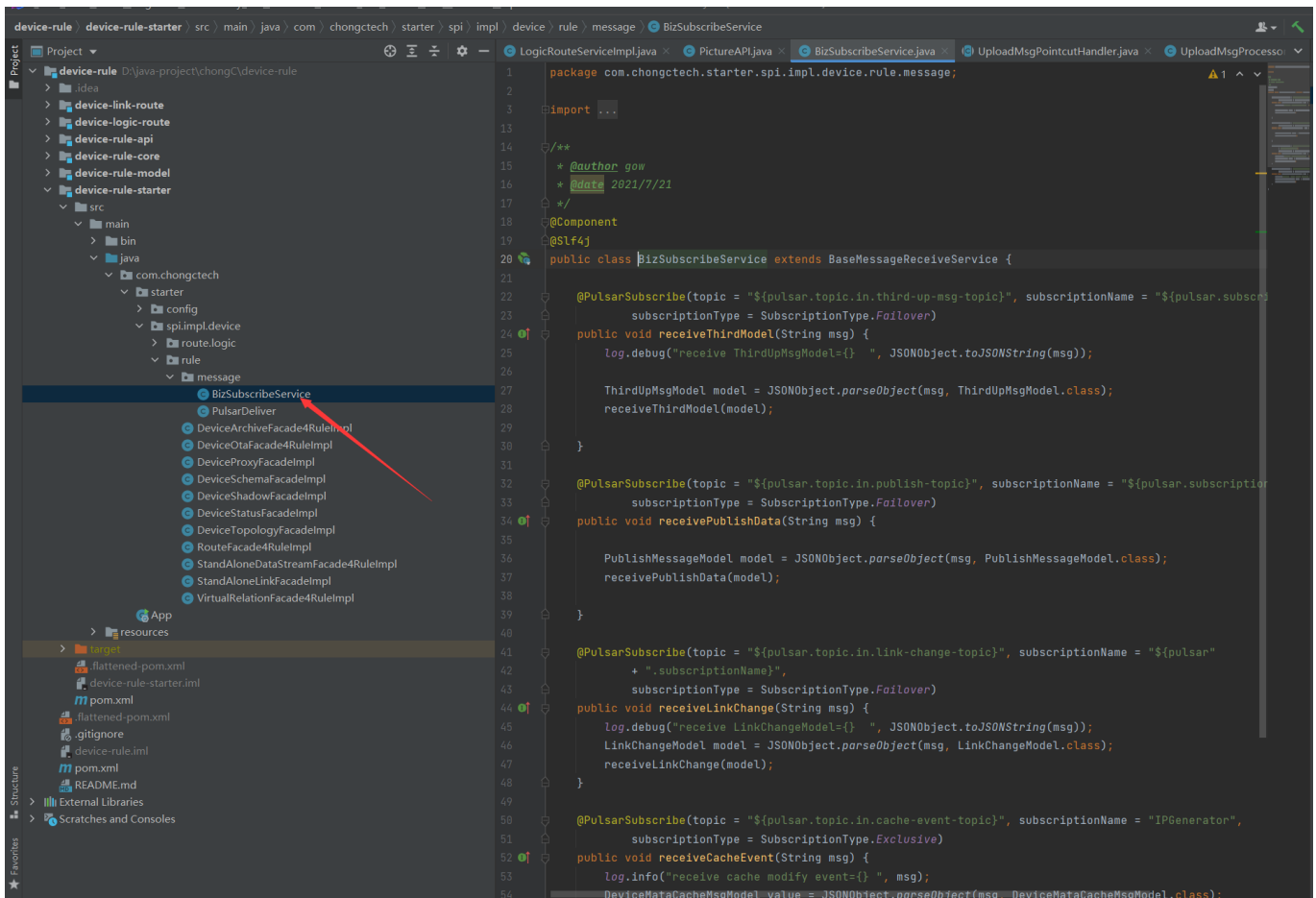
路由实现：包括链路路由，主要实现了设备路由表的新增，删除和查询；逻辑路由主要是查询平台代理关系路由实现

starter模块为spi实现和puslar消费消息，核心包进行业务处理

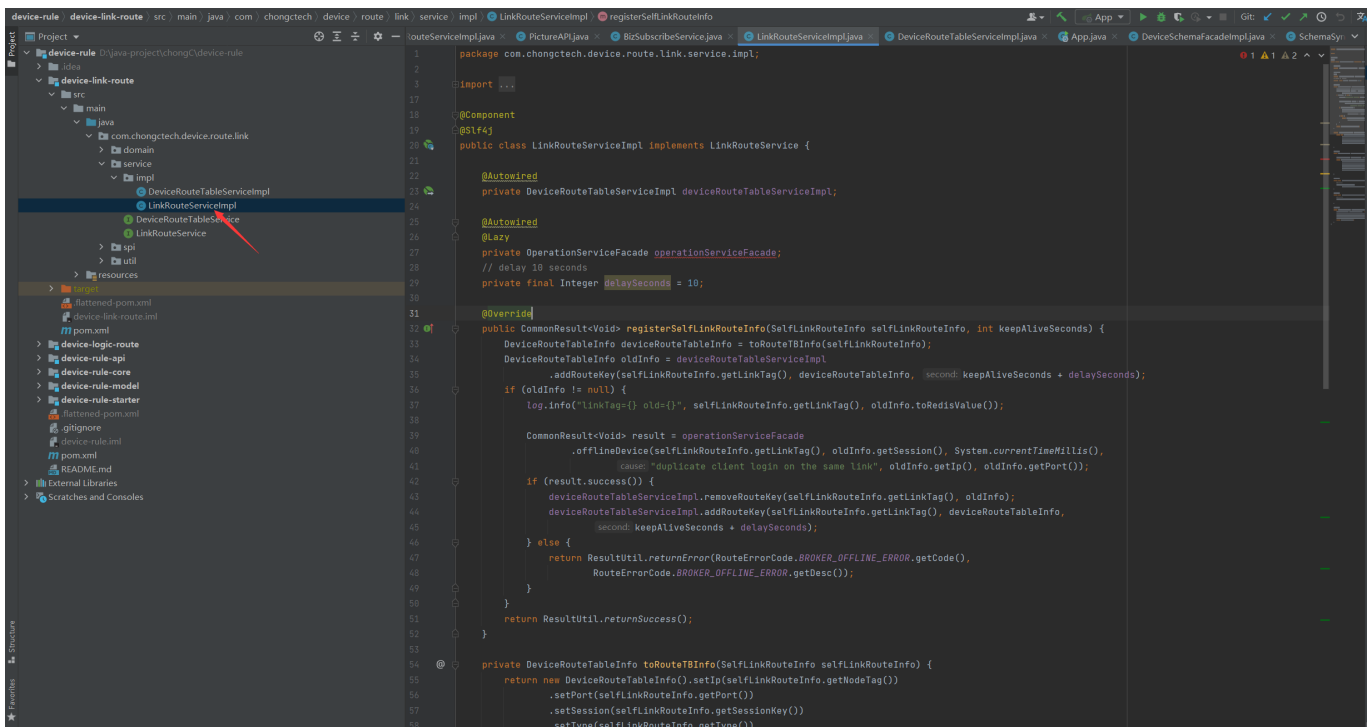
api 入口



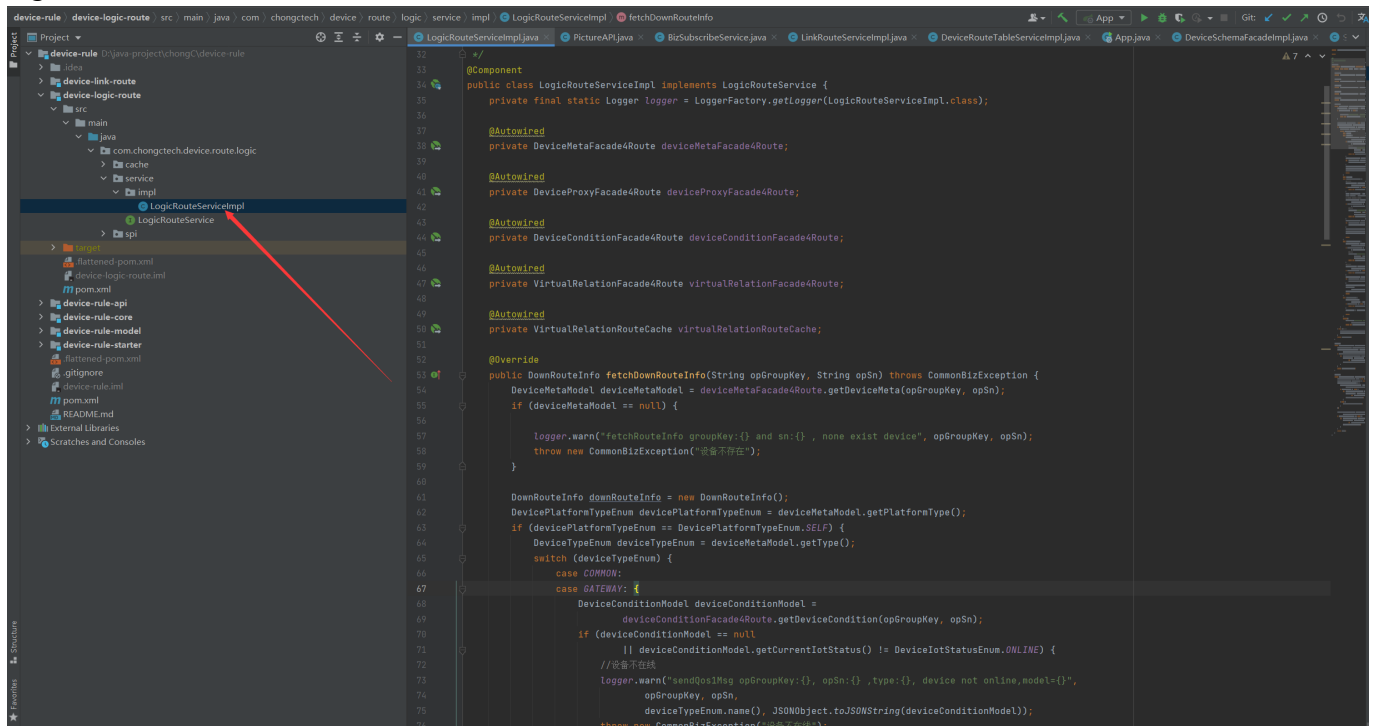
事件入口



link route 入口



logic route 入口

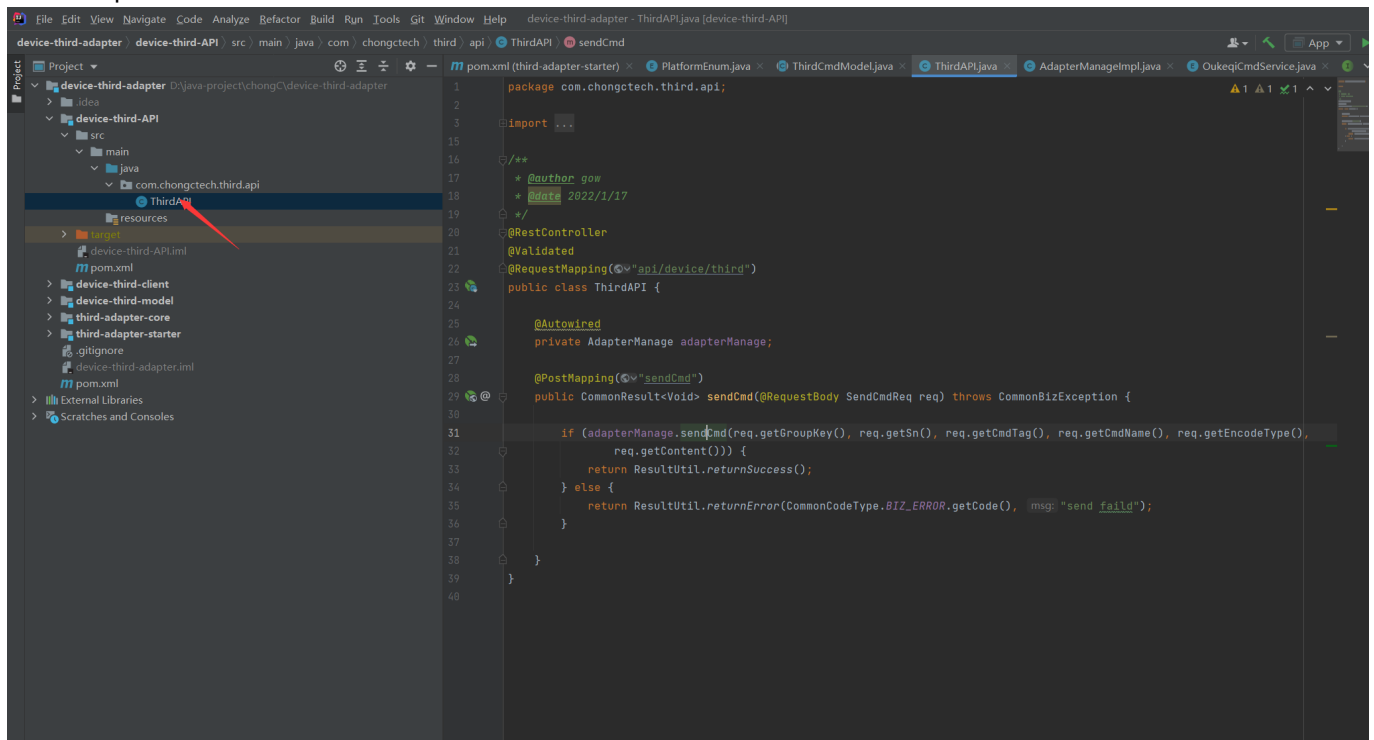


device-third-adapter

该服务主要实现了三方命令的适配接口：

ThirdAPI： 提供了发送命令的接口，其主要实现了通过读取平台三方配置，获取对应平台对应命令的适配器进行命令下发，
发送完成后对设备命令状态进行更新处理。

cmd adapter 入口

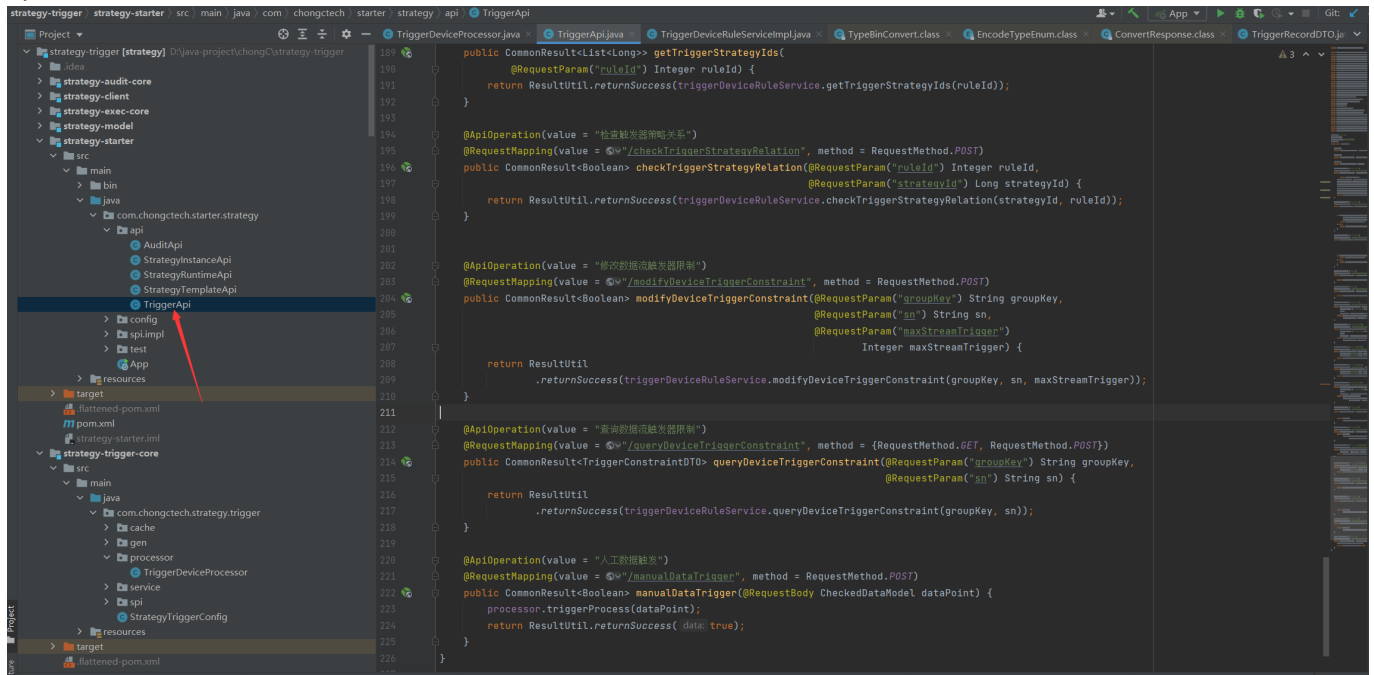


strategy-trigger

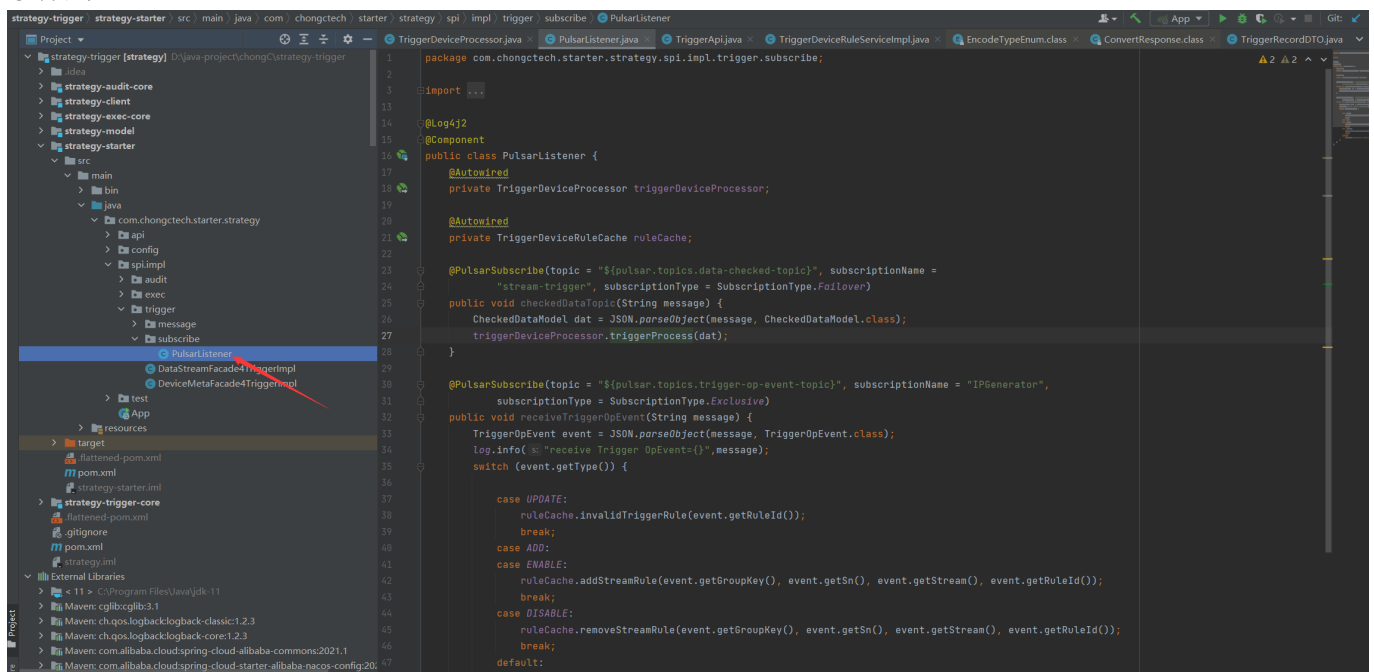
该服务主要实现了触发器模块，其主要提供了触发器增删改查接口和数据触发检验和数据转发；
TriggerApi：提供了添加触发器接口，更新触发器接口，删除触发器接口，禁用、启用触发器接口，
查询触发器接口，
添加，删除和查询触发器策略关系接口，修改和查询触发器限制接口

设备数据触发：消费pulsar设备数据，通过查询触发器规则，进行规则校验，通过规则则将数据触发到下游消费，主要是策略服务消费。

api 入口



事件入口



subscribe service

该服务主要功能为订阅平台数据（数据,命令,日志）进行订阅路由，同时需要维护内部路由表和全局路由表和订阅路由重置API。

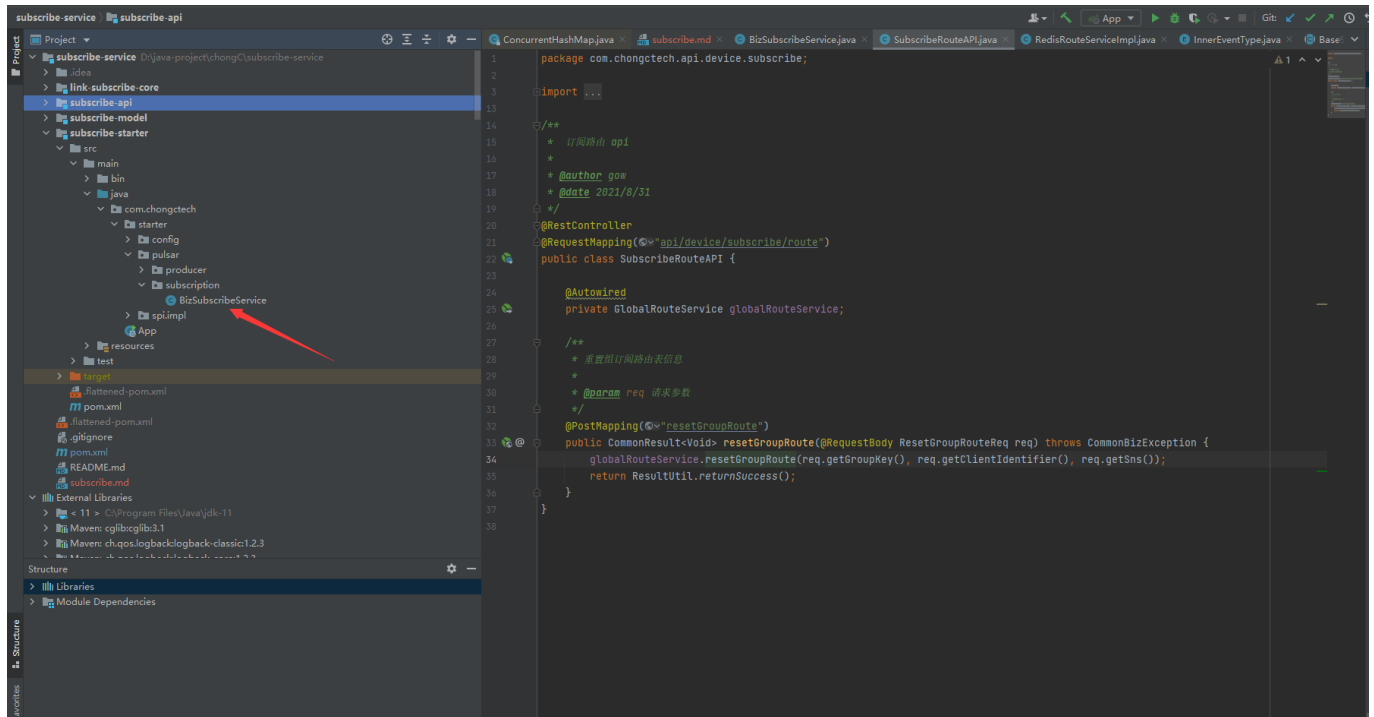
订阅数据：首先是通过pulsar消费数据，然后查询路由信息，根据路由信息进行消息事件pulsar路由；所有的事件通过pulsar触发

维护路由信息，主要通过订阅设备上下线事件进行路由配置拉取，构建本地和全局路由信息，当设备再云端进行路由配置修改时，

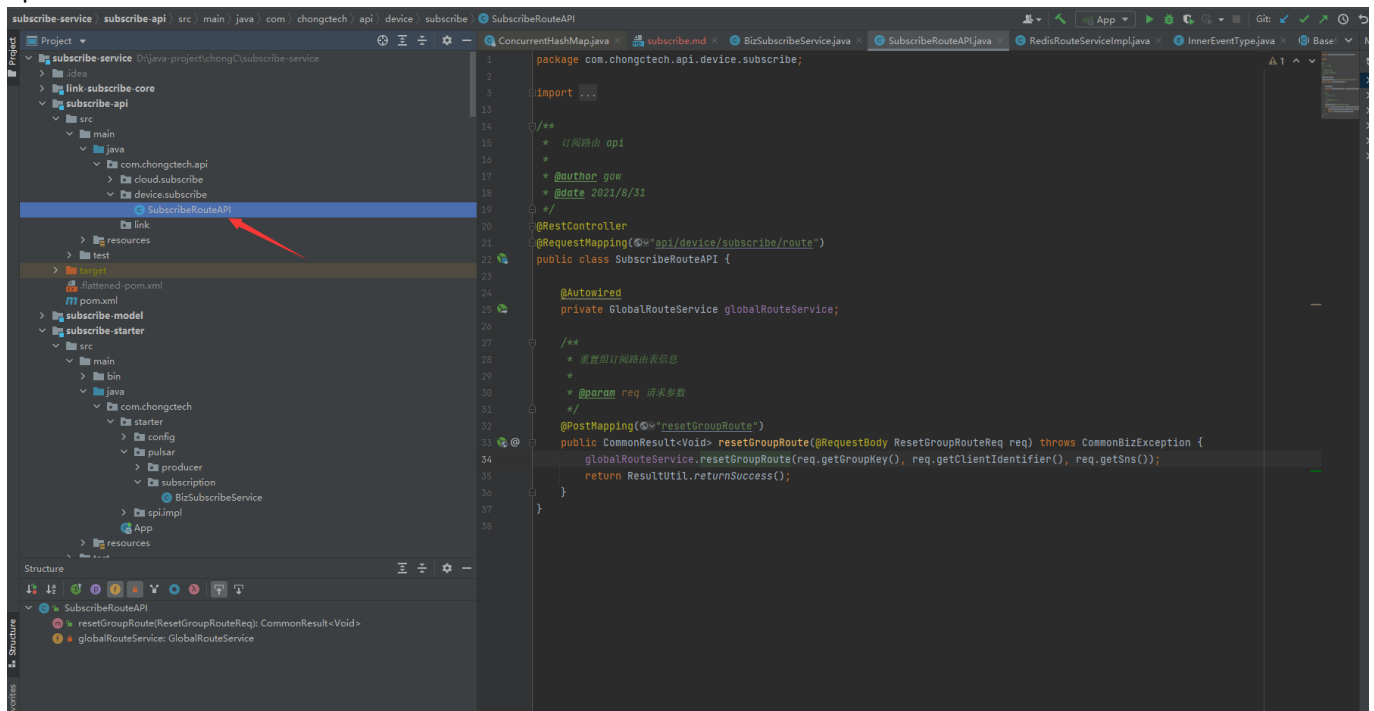
事件通知更新到本地路由和全局路由。所有的事件通过pulsar触发

SubscribeRouteAPI：提供了重置组路由表信息接口，主要是更新本地和全局路由信息

订阅事件入口



api 入口



gospip

主要实现的是sip协议register,ack,bye和message报文实现设备注册和路由表记录，同时提供了invite和bye rest API。

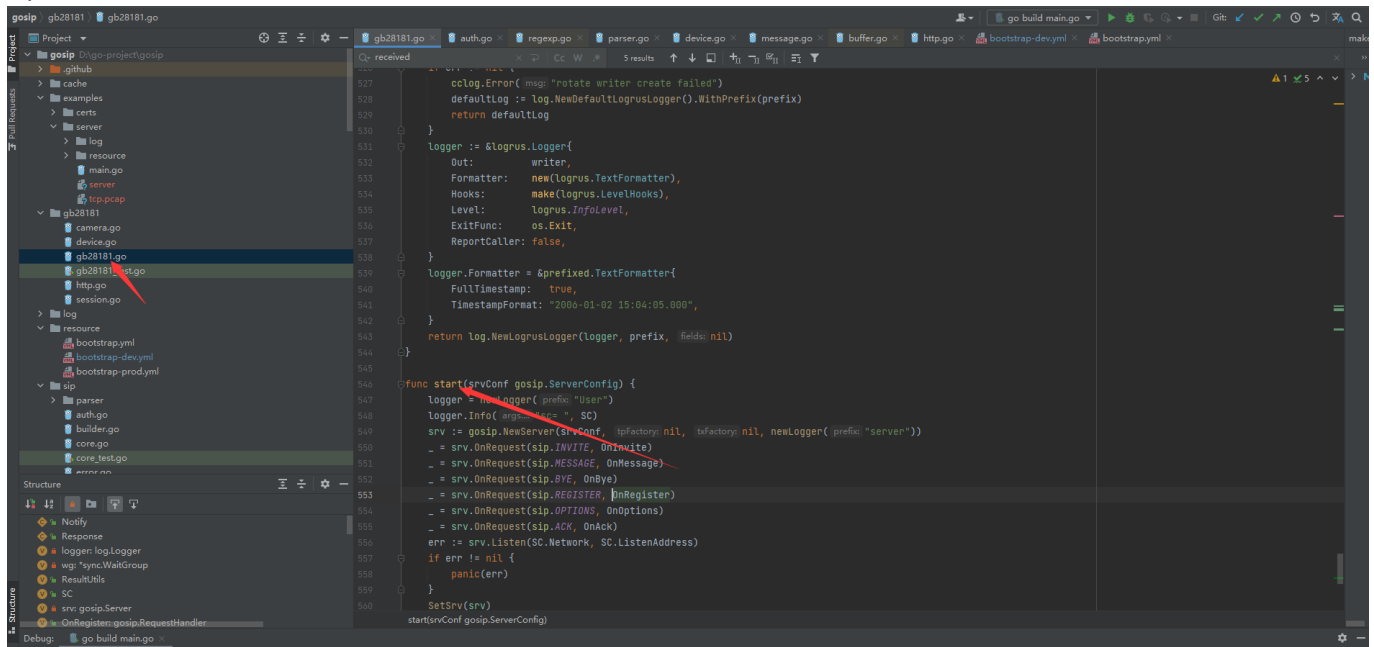
在register报文中，首先是判断是否存在鉴权信息，存在则查询设备平台信息进行鉴权比较，通过则记录本地和全局路由信息；ack 报文主要是判断设备是否存在；

bye报文主要是清理路由信息，message报文主要是更新设备基础信息；

rest api: 主要提供了查询session,invite和bye接口；invite接口主要是检查设备通道，发送bye报文，然后获取srs创建通道，然后构建invite报文并发送报文。发送成功后缓存invite信息。

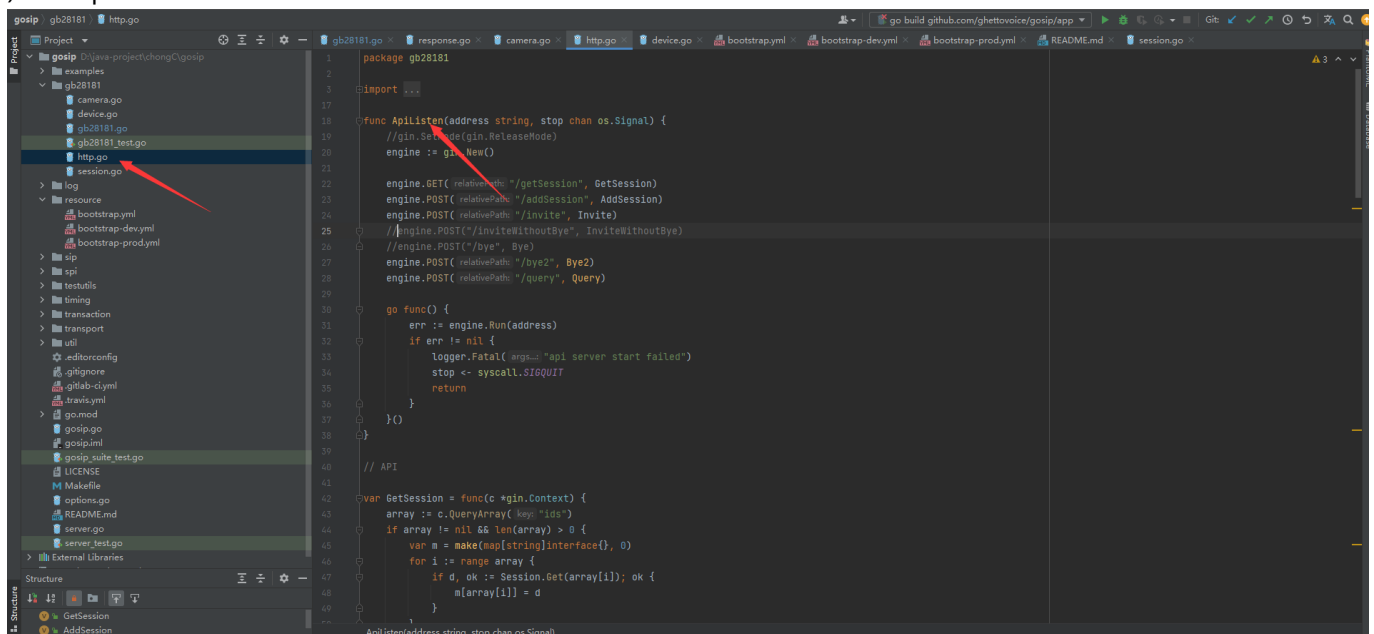
bye接口主要是检查设备通道，判断通道信息是否存在，然后构建bye报文，发送bye报文。

sip 消息处理入口



```
127 // clog.Error(msg: "rotate writer create failed")
128 defaultLog := log.NewDefaultLogger().WithPrefix(prefix)
129 return defaultLog
130
131 logger := &logrus.Logger{
132     Out: writer,
133     Formatter: new(logrus.TextFormatter),
134     Hooks: make(logrus.LevelHooks),
135     Level: logrus.InfoLevel,
136     ExitFunc: os.Exit,
137     ReportCaller: false,
138 }
139 logger.Formatter = &prefixed.TextFormatter{
140     FullTimestamp: true,
141     TimestampFormat: "2006-01-02 15:04:05.000",
142 }
143 return log.NewLogrusLogger(logger, prefix, fields: nil)
144
145 func start(srvConf gospip.ServerConfig) {
146     logger := Logger{prefix: "User"}
147     logger.Infof("start sip server")
148     srv := gospip.NewServer(srvConf, &factory: nil, &factory: nil, newLogger{prefix: "server"})
149     _ = srv.OnRequest(sip.INVITE, OnInvite)
150     _ = srv.OnRequest(sip.MESSAGE, OnMessage)
151     _ = srv.OnRequest(sip.BYE, OnBye)
152     _ = srv.OnRequest(sip.REGISTER, OnRegister)
153     _ = srv.OnRequest(sip.OPTIONS, OnOptions)
154     _ = srv.OnRequest(sip.ACK, OnAck)
155     err := srv.Listen(SC.Network, SC.ListenAddress)
156     if err != nil {
157         panic(err)
158     }
159     SetSrv(srv)
160     start(srvConf gospip.ServerConfig)
```

) rest api入口



```
1 package gb28181
2
3 import (
4     "net/http"
5     "os"
6     "strings"
7 )
8
9 func ApiListen(address string, stop chan os.Signal) {
10     // gin mode
11     engine := gin.New()
12
13     engine.GET("/getSession", GetSession)
14     engine.POST("/addSession", AddSession)
15     engine.POST("/invite", Invite)
16     engine.POST("/inviteWithoutBye", InviteWithoutBye)
17     engine.POST("/bye", Bye)
18     engine.POST("/bye2", Bye2)
19     engine.POST("/query", Query)
20
21 go func() {
22     err := engine.Run(address)
23     if err != nil {
24         logger.Fatalf("api server start failed")
25         stop <- syscall.SIGQUIT
26         return
27     }
28 }()
29
30 // API
31
32 var GetSession = func(c *gin.Context) {
33     array := c.QueryArray("key:ids")
34     if array != nil && len(array) > 0 {
35         var m = make(map[string]interface{}), 0)
36         for i := range array {
37             if d, ok := Session.Get(array[i]); ok {
38                 m[array[i]] = d
39             }
40         }
41     }
42 }
43
44 ApiListen(address string, stop chan os.Signal)
```