GOPHER CHINA 2020

中国 上海 / 2020-11.21-22

华为云的Go语言云原生实践

定义云应用开发的“通信协议”

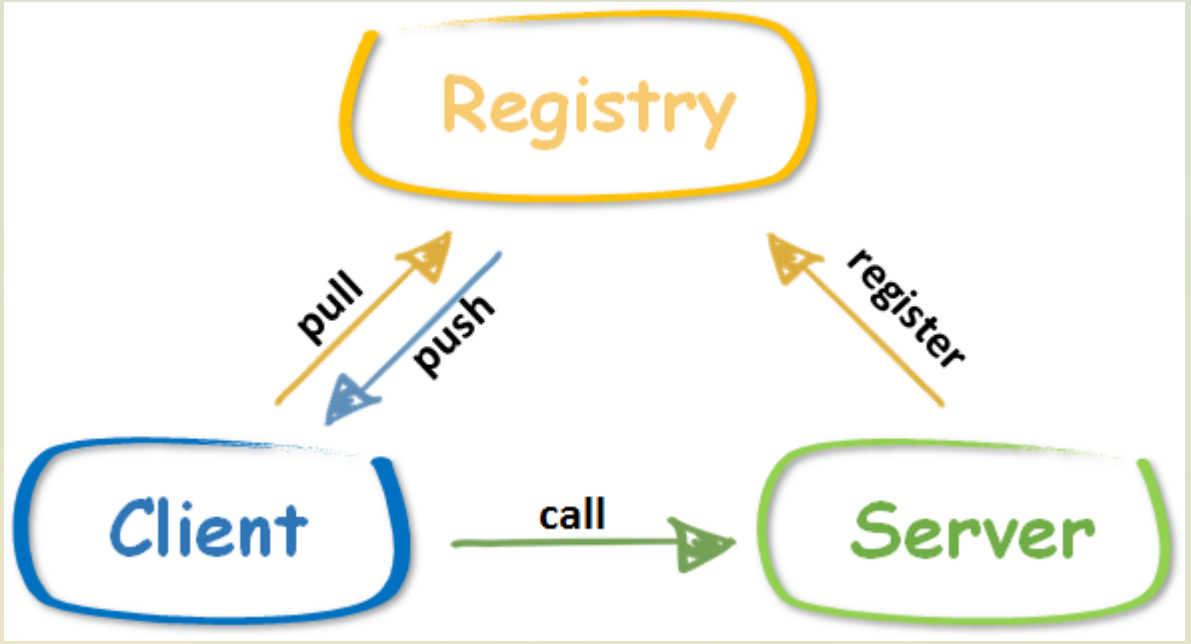
个人介绍

- 11年研发经验，8年云计算领域开发经验
- DevOps，APM，PaaS，混合云等均有深入实践
- 曾在三星负责公司Svoice，AI服务的云化落地
- 现任华为云微服务首席架构师，负责微服务相关产品落地

Agenda

- 展开一个云上应用来看他的设计与实现
- 如何提升研发效能，更快的交付新的云服务

团队的第一go程序：Service Center



然而不只是注册发现

静态与动态信息定义

这里存了什么

```
type MicroService struct {
    ServiceId     string
    AppId         string
    ServiceName   string
    Version       string
    Description   string
    Level         string
    Schemas       []string
    Paths         []*ServicePath
    Status        string
    Properties    map[string]string
    Timestamp     string
    Providers     []*MicroServiceKey
    Alias         string
    LBStrategy    map[string]string
    ModTimestamp  string
    Environment   string
    RegisterBy    string
    Framework     *FrameWorkProperty
}
```

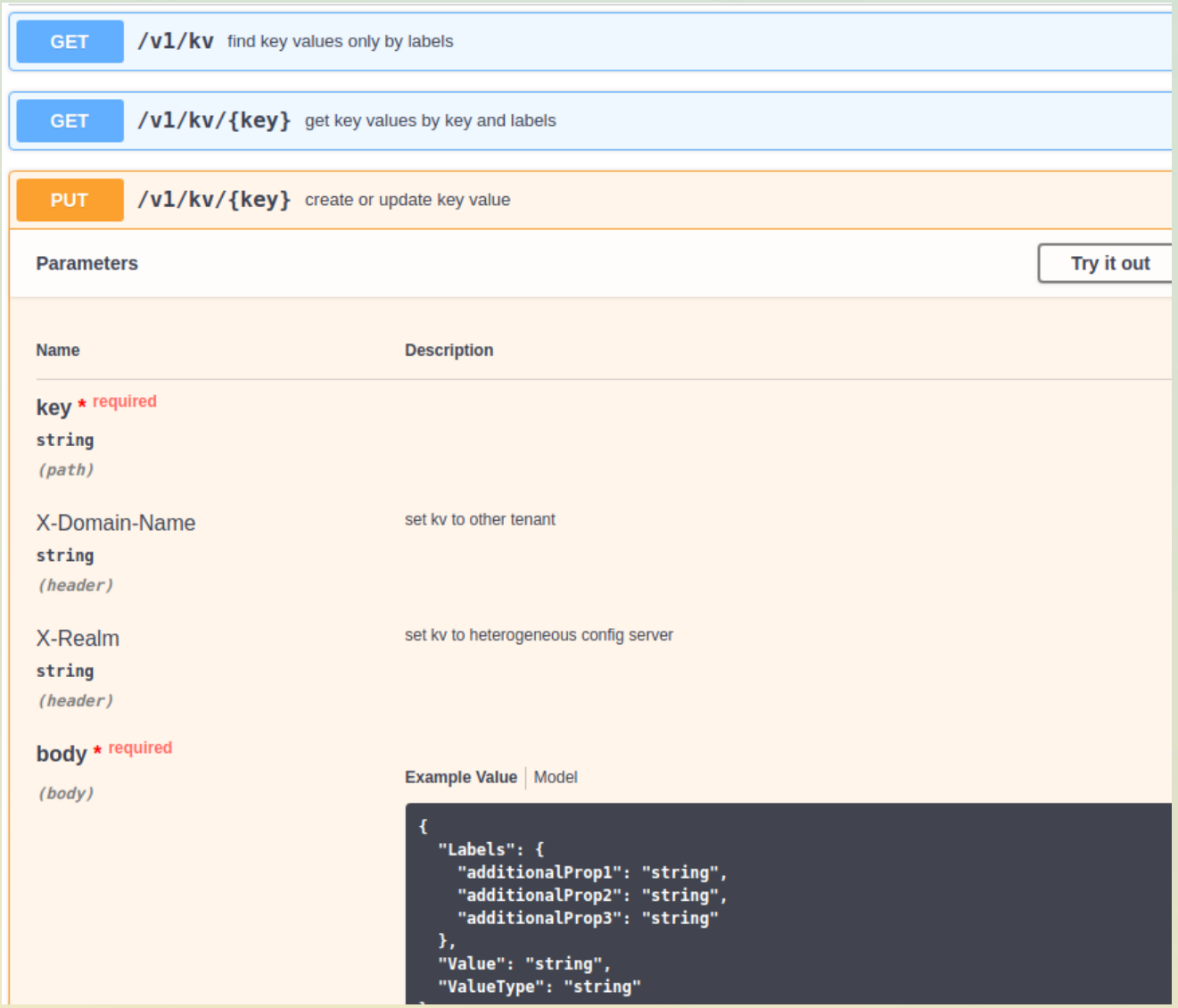
1:n

```
type MicroServiceInstance struct {
    InstanceId      string
    ServiceId       string
    Endpoints       []string
    HostName        string
    Status          string
    Properties      map[string]string
    HealthCheck     *HealthCheck
    Timestamp       string
    DataCenterInfo  *DataCenterInfo
    ModTimestamp    string
    Version         string
}
```

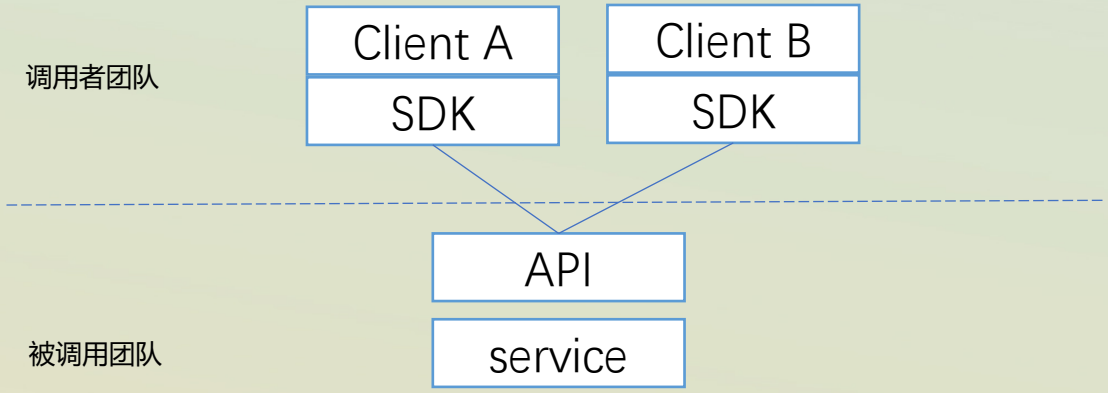
- 冗余度降低
- 减少网络压力

契约管理

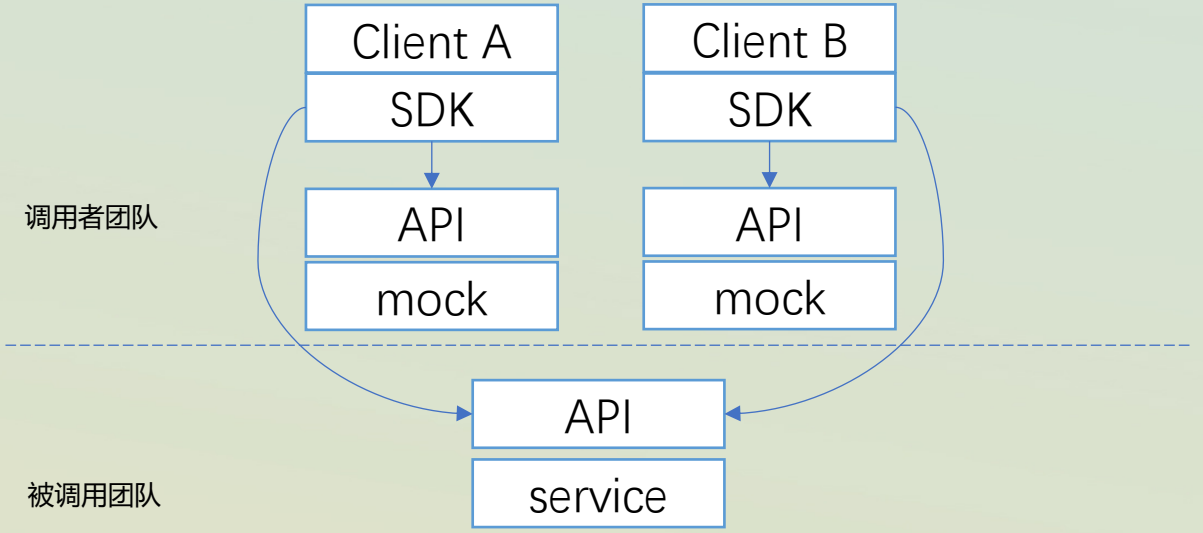
```go
//URLPatterns defined config operations
func (r *KVResource) URLPatterns() []restful.Route {
    return []restful.Route{
        {
            Method:           http.MethodPut,
            Path:             "/v1/kv/{key}",
            ResourceFuncName: "Put",
            FuncDesc:         "create or update key value",
            Parameters: []*restful.Parameters{
                {
                    DataType:  "string",
                    Name:      "key",
                    ParamType: goRestful.PathParameterKind,
                }, {
                    DataType:  "string",
                    Name:      "X-Domain-Name",
                    ParamType: goRestful.HeaderParameterKind,
                    Desc:      "set kv to other tenant",
                }, {
                    DataType:  "string",
                    Name:      "X-Realm",
                    ParamType: goRestful.HeaderParameterKind,
                    Desc:      "set kv to heterogeneous config server",
                },
            },
            Returns: []*restful.Returns{
                {
                    Code:    http.StatusOK,
                    Message: "true",
                },
            },
            Consumes: []string{"application/json"},
            Produces: []string{"application/json"},
            Read:     &KVBody{},
        }, {
            Method:           http.MethodGet,
            Path:             "/v1/kv/{key}",
            ResourceFuncName: "Find",
            FuncDesc:         "get key values by key and labels",
            Parameters: []*restful.Parameters{
                {
                    DataType:  "string",
```



为什么应用API first --- 效率低下的开发模式



客户端总是在等待API 服务真正的发布，甚至是部署



通过注册到Service Center的API生成mock代码进行集成测试。等到服务发布后，再切换到真实服务进行测试

为什么应用API first --- 糟糕的API设计

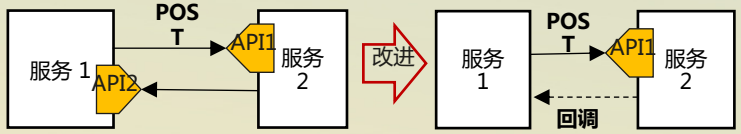
- 不符合restful风格
- 字段风格不一致
- 扩展能力差
- 易用性差
- API能力相似

服务依赖管理

是否双向依赖

如下两种均属于**循环调用**，循环调用违背了服务设计时对层次的要求，应由上层服务调用下层服务，同时循环调用也增加了服务间的耦合度，这是一种错误的设计。

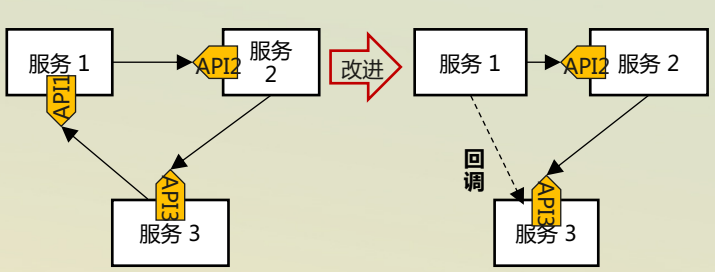
循环依赖的情况，识别出形成循环依赖的服务后，可以采用**webhook**的集成方式进行解耦



是否循环依赖

如下两种均属于**循环调用**，循环调用违背了服务设计时对层次的要求，应由上层服务调用下层服务，同时循环调用也增加了服务间的耦合度，这是一种错误的设计。

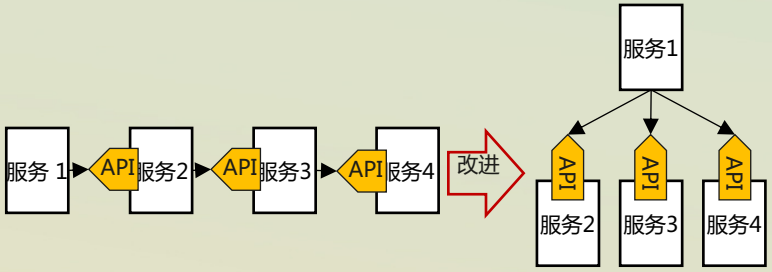
如下两种循环依赖的情况，识别出形成循环依赖的服务后，可以采用**webhook**的集成方式进行解耦



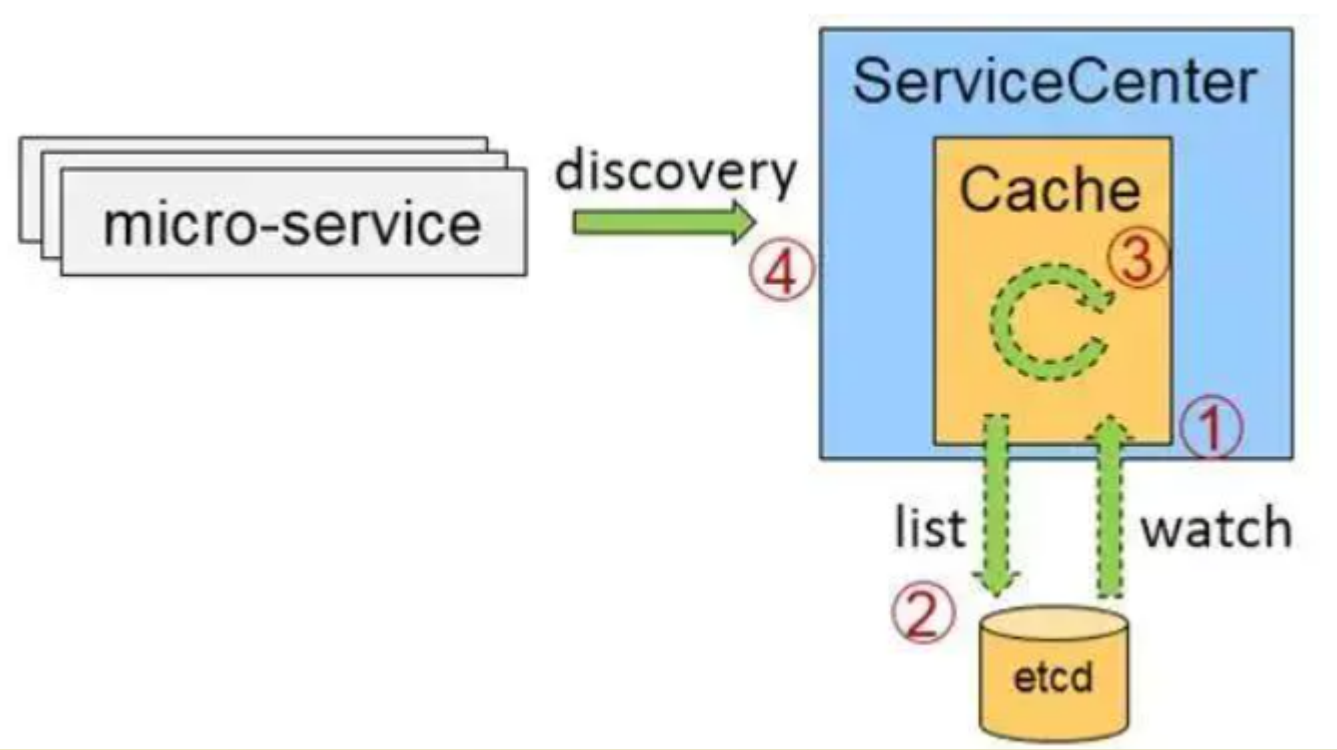
调用最长深度，调用API总个数

服务调用形成**链条依赖**，深度过深即增加了服务间的耦合度，也使得数据丢失的风险大大增加。

对这种类型的调用关系进行解耦，首先考虑服务的**层级关系**，以上层服务向下层服务调用的方式进行解耦



缓存机制

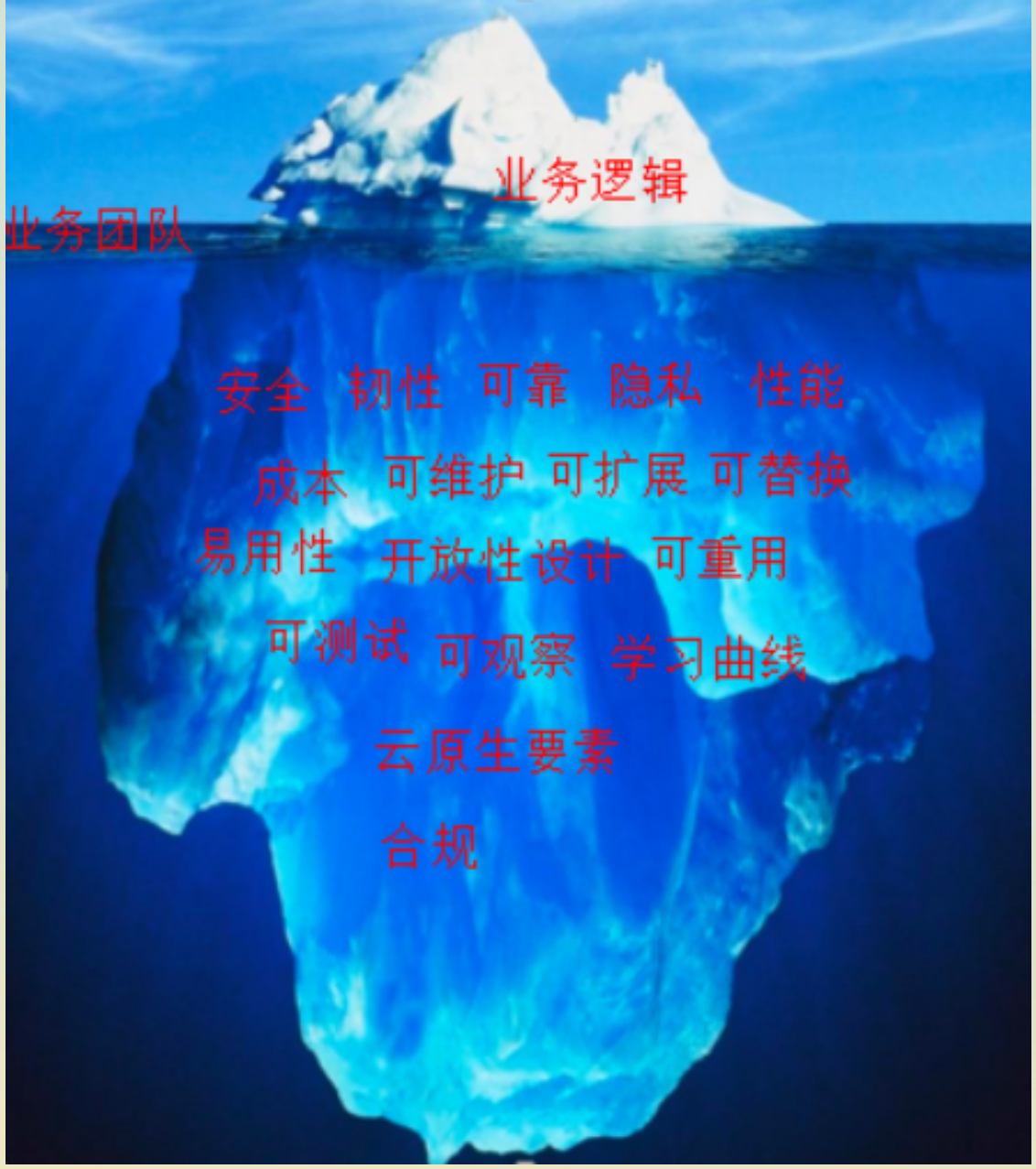

micro-service
discovery
4
ServiceCenter
Cache
3
1
list
watch
2
etcd

注册发现增强的收益

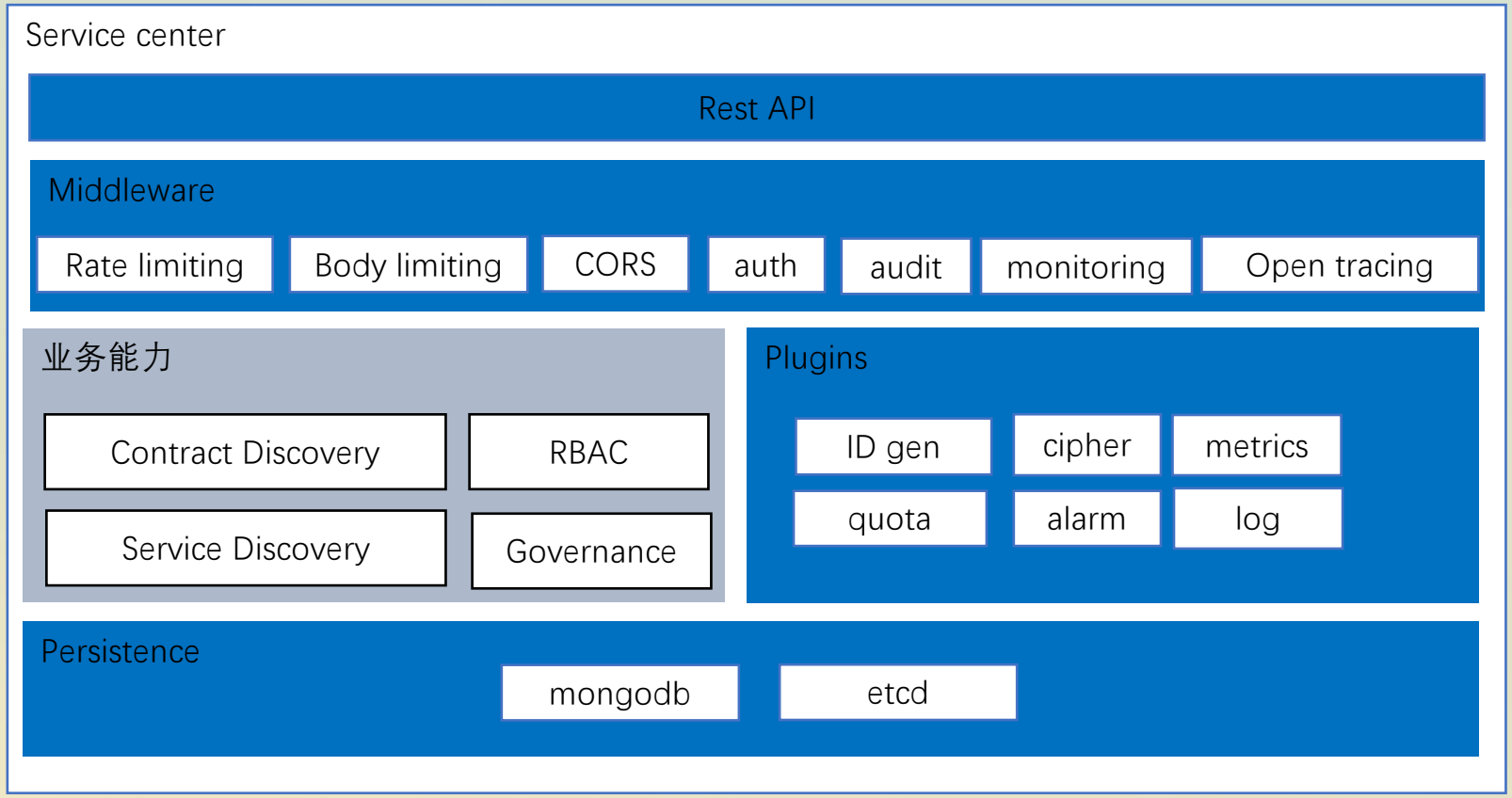
- 为架构师提供一个审视视角，比如合理的API设计，依赖关系
- 规范化微服务元数据，防止滥用，减少冗余数据，降低网络开销

然而远远不止交付业务功能

冰山之下


业务逻辑
业务团队
安全 韧性 可靠 隐私 性能
成本 可维护 可扩展 可替换
易用性 开放性设计 可重用
可测试 可观察 学习曲线
云原生要素
合规

Service Center的层叠架构

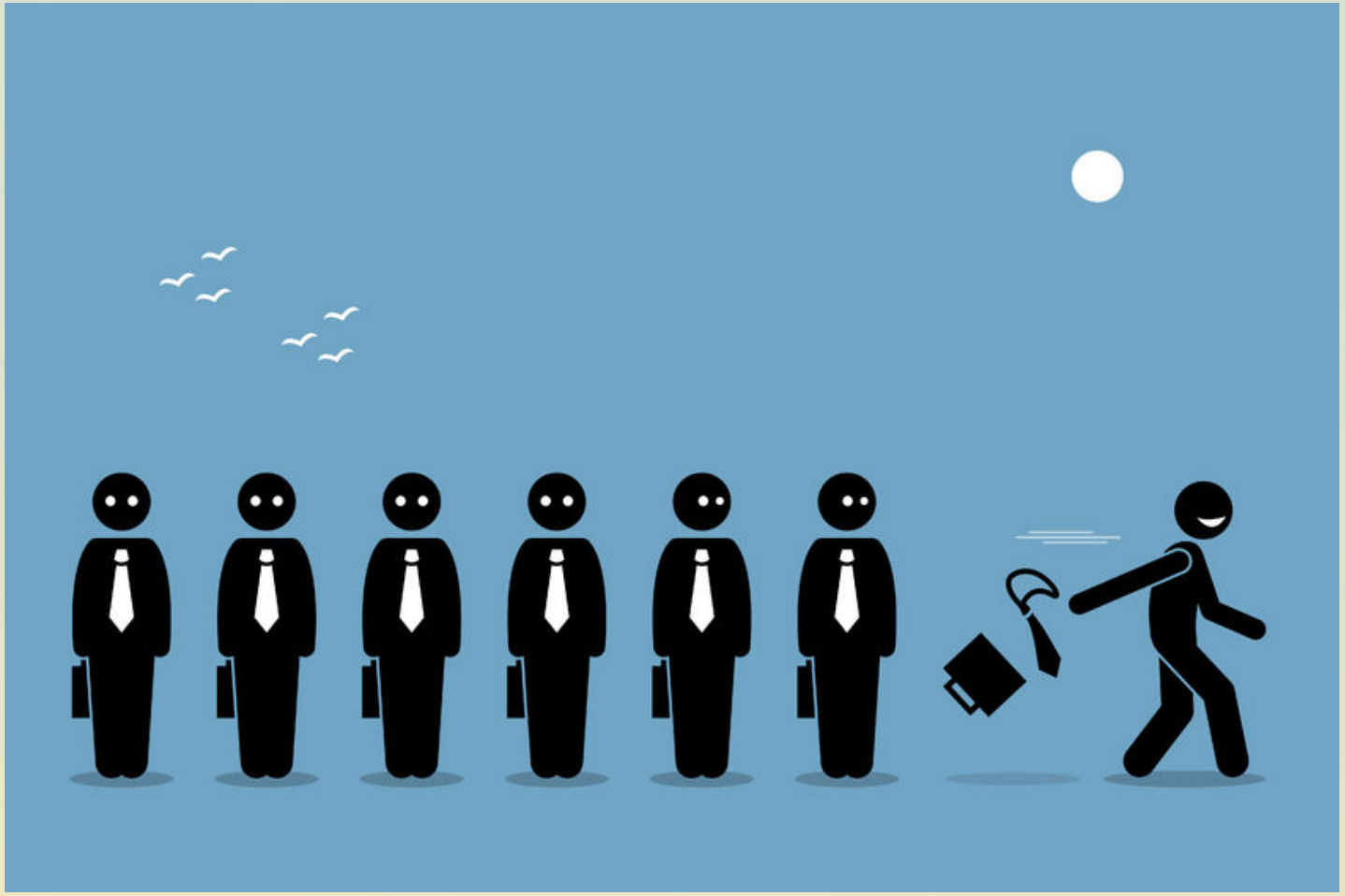


四个主要的模块：

- **服务注册发现：**通过注册发现完成服务拓扑的感知
- **契约发现：**每个服务具备一个契约记录，支持多种格式如Open API，gRPC proto
- **RBAC：**基于角色的访问控制，管理员可以管理账号，将账号分发给微服务或者不同人员
- **服务治理：**针对微服务下发治理规则，比如重试，限流，熔断，路由策略等。

https://github.com/apache/servicecomb-service-center

不同交付局点，不同代码分支交付

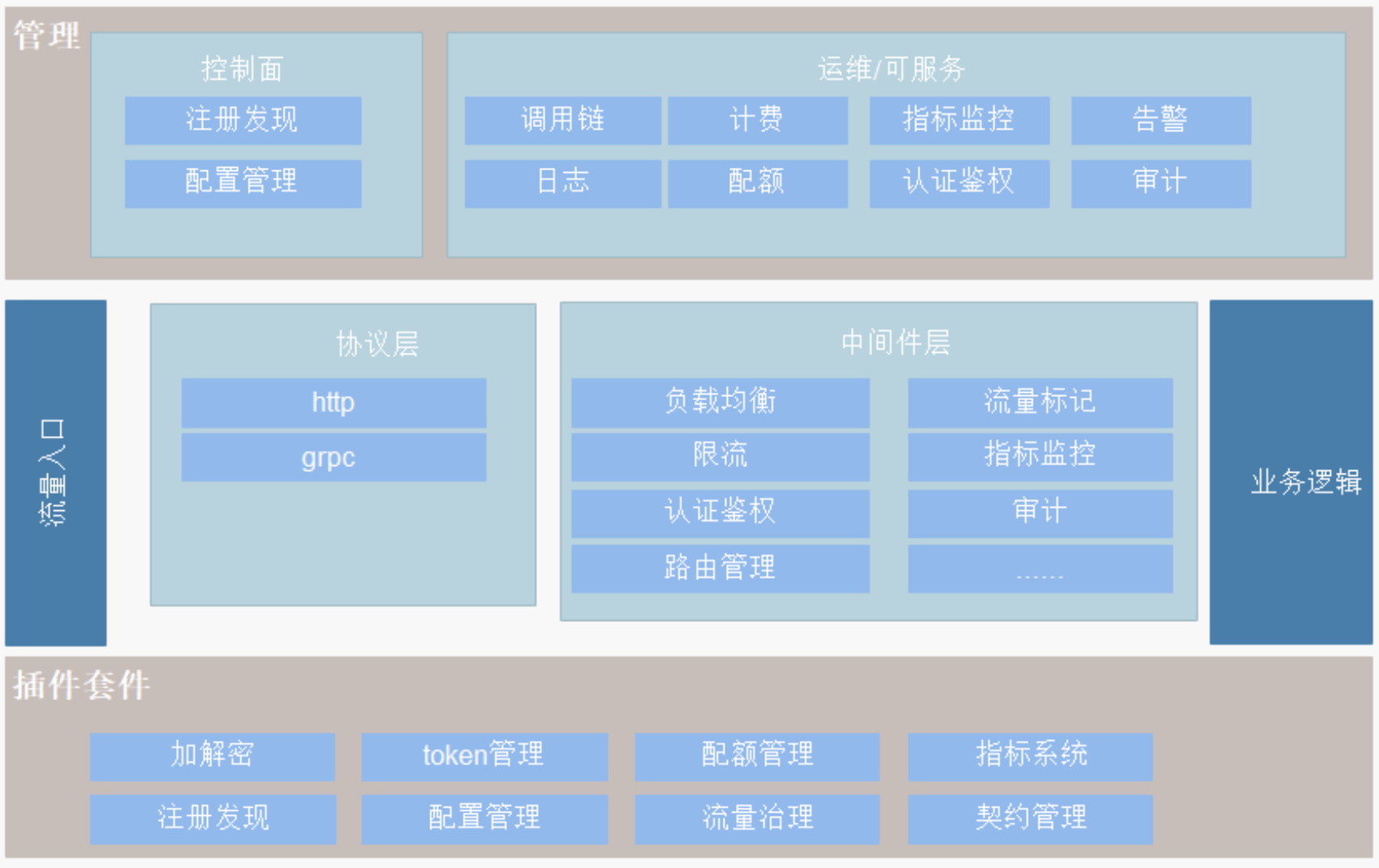


停止服务

如何面对

- **可插拔：**按需在编译期引入
- **异构服务：**一个后台服务可能有多种具体实现。
- **不同的算法：**解密工具，ID生成器，面对不同的交付场景或者是安全要求，都要通过不同实现来替换算法。
- **分布系统难以治理：**框架可以帮助满足云原生应用，如何实现这样的框架

Go chassis云应用开发框架



https://github.com/go-chassis/go-chassis

手段1：将后端服务作为插件使用

常见的后端服务

- 配额管理
- 认证鉴权服务
- 对象存储服务

面向用户

Implement and Install

```go
type inMemory struct {
}

func newQuotaManager(options quota.Options) (quota.Manager, error) {
    return &inMemory{}, nil
}
func init() {
    quota.Install( name: "mock", newQuotaManager)
}
```

Import package and Configure

```yaml
servicecomb:
  quota:
    plugin: mock
```

Call

```go
quota.PreCreate("some cloud service", "some user", "cpu", 2)
```

如何实现插件机制

1. 定义后端接口

```
73    //Manager could be a quota management system
74    type Manager interface {
75        GetQuotas(service, domain string) ([]*Quota, error)
76        IncreaseUsed(service, domain, resource string, used int64) error
77        DecreaseUsed(service, domain, resource string, used int64) error
78    }
```

2. 定义插件集，提供安装API

```
34    type newManager func(opts Options) (Manager, error)
35
36    var plugins = make(map[string]newManager)
37
38    //Install install quota plugin
39    func Install(name string, f newManager) {
40        plugins[name] = f
41    }
```

模块初始化

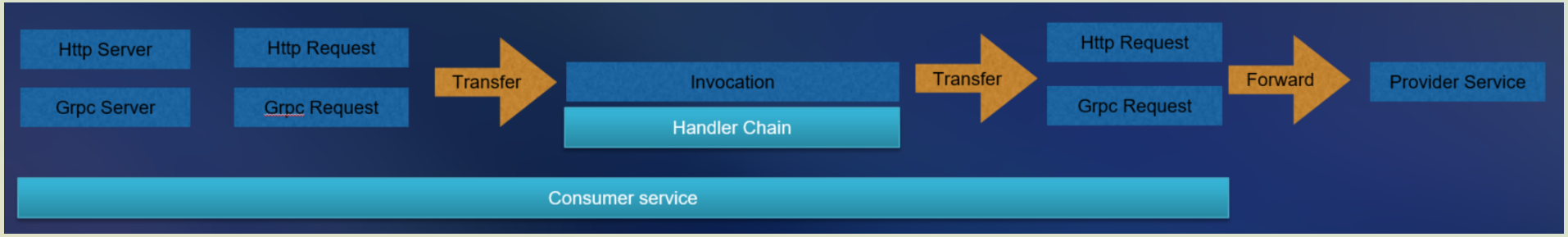
```
44  func Init(opts Options) error {
45      if opts.Plugin == "" : nil ↵
48
49      f, ok := plugins[opts.Plugin]
50      if !ok {
51          return fmt.Errorf( format: "not supported [%s]", opts.Plugin)
52      }
53      var err error
54      defaultManager, err = f(opts)
55      if err != nil {
56          return err
57      }
58      openlog.Info(fmt.Sprintf( format: "quota management system [%s@%s] enabled", opts.Plugin, opts.Endpoint))
59      return nil
60  }
```

手段2：沉淀需求基线

基线需求

- 请求体必须做大小限制
- API必须限流
- 密码不能明文存储
- 访问进行认证鉴权
- 无单点故障
- 访问审计
- 运维能力

标准化运行时模型



不同部门可能有私有协议诉求，那么服务治理就交给核心框架完成。协议由业务部门决定自主研发或是集成现有协议。当你发现公司内部不同部门都在开发自己的协议做自己的服务治理时，再向将业务统一一个架构，一个工具链上，将非常困难。

Handler chain

```go
type accessLog struct {
    record func(time.Time, *invocation.Invocation)
}

// Handle ...
func (a *accessLog) Handle(chain *handler.Chain, i *invocation.Invocation, cb invocation.ResponseCallBack) {
    now := time.Now()
    chain.Next(i, func(response *invocation.Response) {
        cb(response)
        a.record(now, i)
    })
}
```

- Middleware
  - access log
  - Basic Auth
  - Circuit breaker
  - JWT
  - Monitoring
  - Rate limiting v1
  - Rate limiting v2

- 认证鉴权，监控，限流，调用链追踪等与流量相关的能力

声明式使用

1. Import package

```
import _ github.com/go-chassis/go-chassis/v2/middleware/ratelimiter
```

2. Configure handler chain

```
servicecomb:
  handler:
    chain:
      Provider:
        default: traffic-marker,rate-limiter
```

3. Clarify Policy

```
servicecomb:
  match:
    traffic-to-some-api-from-jack: |
        matches:
          - headers:
              cookie:
                regex: "^(.*?;)?(user=jack)(;.*)?$"
              os:
                contains: linux
            apiPath:
              exact: "/some/api"
            method:
              - GET
              - POST
            trafficMarkPolicy: once
  rateLimiting:
    limiterPolicy1: |
      match: traffic-to-some-api-from-jack
      rate: 10
      burst: 1
    nonMarked: |
      match: none
      rate: 100
      burst: 1
```

命令式调用

```go
//Registry holds all of metrics collectors
//name is a unique ID for different type of metrics
type Registry interface {
    CreateGauge(opts GaugeOpts) error
    CreateCounter(opts CounterOpts) error
    CreateSummary(opts SummaryOpts) error
    GaugeSet(name string, val float64, labels map[string]string) error
    CounterAdd(name string, val float64, labels map[string]string) error
    SummaryObserve(name string, val float64, Labels map[string]string) error
}
```

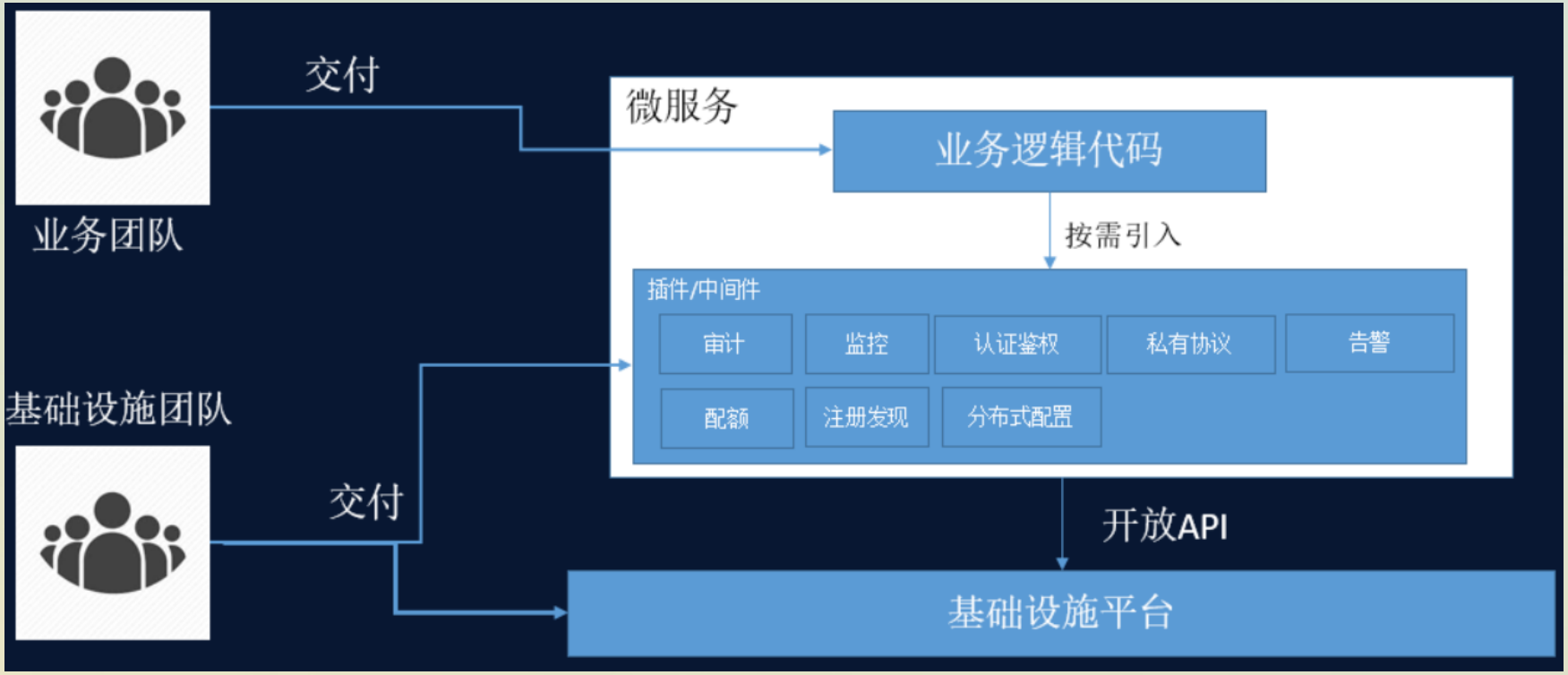
```yaml
servicecomb:
  metrics:
    apiPath: /metrics       # we ca
    enable: true
    enableGoRuntimeMetrics: true
    enableCircuitMetrics: true
```

- 除了通用的，还可上报自定义指标
- 只需配置文件开启即可开启

插件能力全景图


go chassis插件能力
注册发现
service center
kubernetes
动态配置
kie
apollo
可观测
zipkin
prometheus
协议
http
grpc
安全
JWT
cipher
basic auth
韧性
容错
限流
熔断
工具
配额管理
访问审计
流量管理
流量标记
AZ亲和性
金丝雀发布

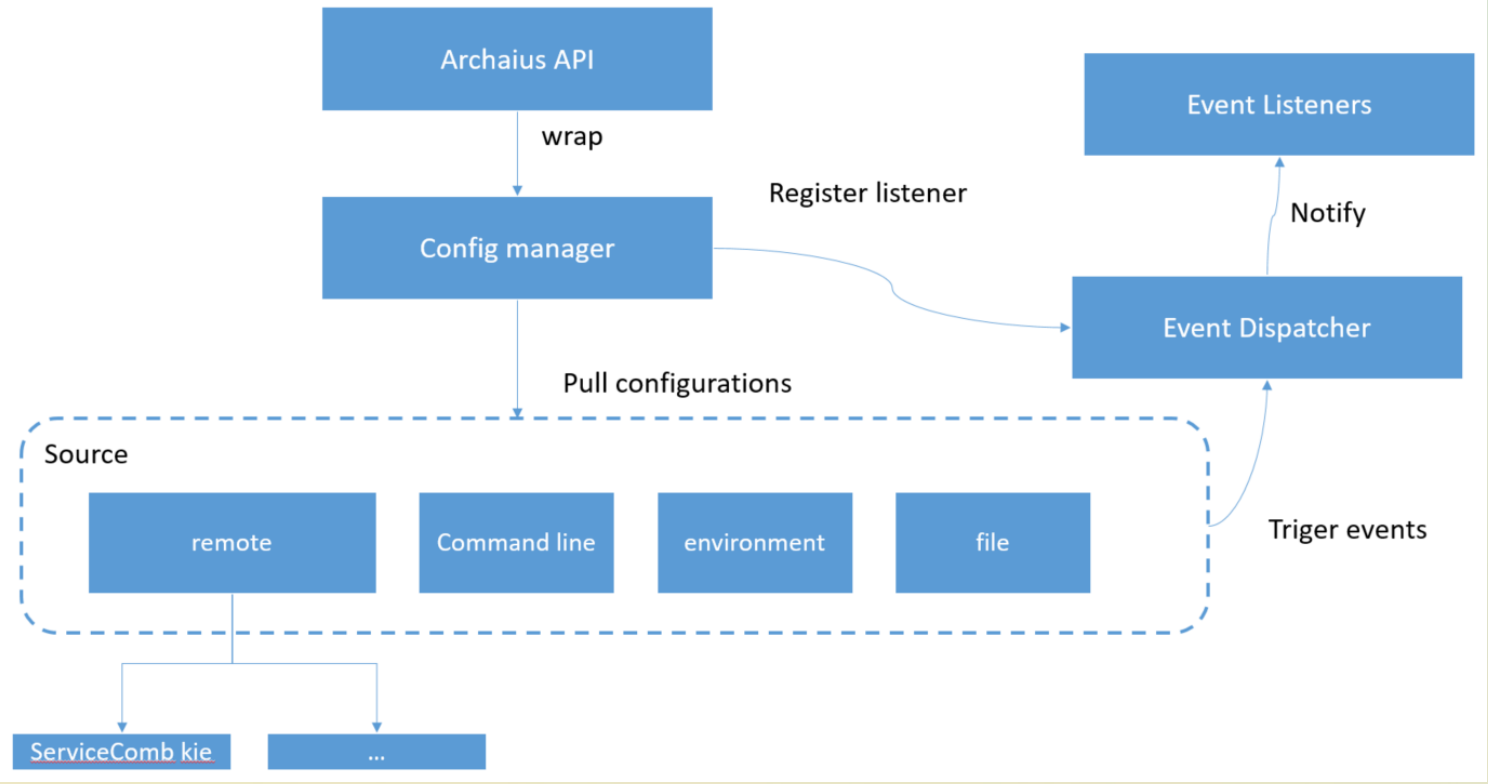
收益



- 对于庞大的系统可以进行mock测试，提升交付质量
- 应对不同的交付场景
- 保证后端可替换性
- 职责界面分离

手段3：配置治理

Go chassis内置配置热加载



- 远端管理：生产环境
- 本地管理：集成测试
- 内存管理：UT测试

Read config files

if you have a yaml config

```
some:
  config: 1
ttl: 30s
service:
  name: ${NAME||go-archaius}
  addr: ${IP||127.0.0.1}:${PORT||80}
```

after adding file

```
archaius.AddFile("/etc/component/xxx.yaml")
```

you can get value

```
ttl := archaius.GetString("ttl", "60s")
i := archaius.GetInt("some.config", "")
serviceName := archaius.GetString("service.name", "")
serviceAddr := archaius.GetString("service.addr", "")
```

手段4：易处理

协议服务管理

```go
// ProtocolServer interface for the protocol server, a server should implement init, register, start, and stop
type ProtocolServer interface {
    //Register a schema of microservice,return unique schema id,you can specify schema id and microservice name of this schema
    Register(interface{}, ...RegisterOption) (string, error)
    Start() error
    Stop() error
    String() string
}
```

this config will launch 2 http server and 1 grpc server

```yaml
servicecomb:
  protocols:
    rest:
      listenAddress: 0.0.0.0:5000
    rest-admin:
      listenAddress: 0.0.0.0:5001
    grpc:
      listenAddress: 0.0.0.0:6000
```

- 统一Server模型
- 通过配置进行协议服务管理

业务代码编写并注册

1. 编写业务

```go
//通常持有一批API，并定义API Patterns
type HelloResource struct {
}

//业务API
func (r *HelloResource) SayHi(b *rf.Context) {
    b.Write([]byte("hello, go chassis"))
    return
}

//定义所有的API Patterns，用于API路由
func (r *HelloResource) URLPatterns() []rf.Route {
    return []rf.Route{
        {Method: http.MethodGet, Path: "/hello", ResourceFunc: r.SayHi},
    }
}
```

2. 注册业务逻辑

```go
chassis.RegisterSchema("rest", &HelloResource{})
```

3. 框架启动

```go
if err := chassis.Init(); err != nil {
    openlog.Fatal("Init failed." + err.Error())
    return
}
chassis.Run()
```

优雅停机过程

```go
func GracefulShutdown(s os.Signal) {
    if !config.GetRegistratorDisable() {
        registry.HBService.Stop()
        openlog.Info( message: "unregister servers ...")
        if err := server.UnRegistrySelfInstances(); err != nil {
            openlog.Warn("servers failed to unregister: " + err.Error())
        }
    }

    for name, s := range server.GetServers() {
        openlog.Info( message: "stopping server " + name + "...")
        err := s.Stop()
        if err != nil {
            openlog.Warn("servers failed to stop: " + err.Error())
        }
        openlog.Info(name + " server stop success")
    }

    openlog.Info( message: "go chassis server gracefully shutdown")
}
```

1. 停止心跳
2. 注销实例
3. 中断端口监听
4. 处理所有请求后退出

自定停机过程

```go
//HijackSignal set signals that want to custom signals.
func HijackSignal(sigs ...os.Signal) {
    goChassis.sigs = sigs
}

//InstallPreShutdown install a function executed before graceful shutdown
func InstallPreShutdown(name string, f func(os.Signal)) {
    // lazy init
    if goChassis.preShutDownFuncs == nil {
        goChassis.preShutDownFuncs = make(map[string]func(os.Signal))
    }
    goChassis.preShutDownFuncs[name] = f
}

//InstallPostShutdown install a function executed after graceful shutdown
func InstallPostShutdown(name string, f func(os.Signal)) {
    // lazy init
    if goChassis.postShutDownFuncs == nil {
        goChassis.postShutDownFuncs = make(map[string]func(os.Signal))
    }
    goChassis.postShutDownFuncs[name] = f
}

//HijackGracefulShutdown replace GracefulShutdown
func HijackGracefulShutdown(f func(os.Signal)) {
    goChassis.hijackGracefulShutdown = f
}
```

- 停机前后可以定制逻辑
- 可以彻底劫持go chassis原本停机过程
- 可以替换默认的系统信号

http的优雅停机实现

```
322  func (r *restfulServer) Stop() error {
323      if r.server == nil {
324          openlog.Info( message: "http server never started")
325          return nil
326      }
327      //only go>=1.8 support graceful shutdown.
328      if err := r.server.Shutdown(context.TODO()); err != nil {
329          openlog.Warn("http shutdown error: " + err.Error())
330          return err
331      }
332      return nil
333  }
```

手段5：轻量级内核

让我们打开go.mod看一看

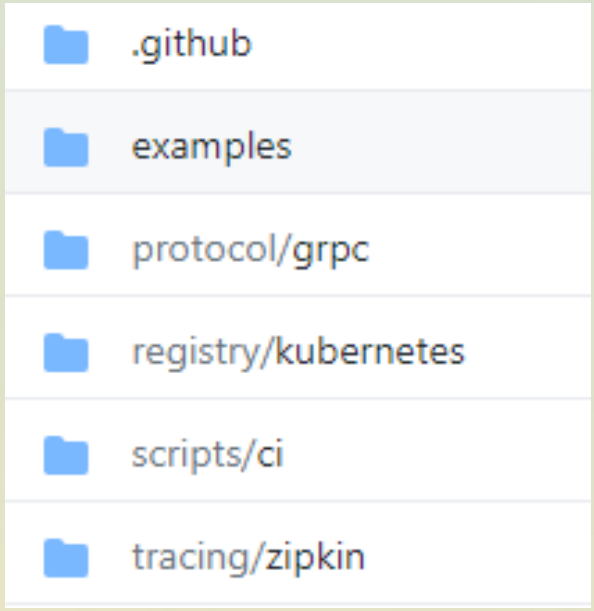
```
module github.com/go-chassis/go-chassis/v2

require (
        github.com/cenkalti/backoff v2.0.0+incompatible
        github.com/dgrijalva/jwt-go v3.2.0+incompatible
        github.com/emicklei/go-restful v2.12.0+incompatible
        github.com/go-chassis/cari v0.0.0-20201113135522-88c21500ca3f
        github.com/go-chassis/foundation v0.1.1-0.20200825060850-b16bf420f7b3
        github.com/go-chassis/go-archaius v1.3.6-0.20201103103813-43dd1680ebfb
        github.com/go-chassis/go-restful-swagger20 v1.0.3-0.20200310030431-17d80f34264f
        github.com/go-chassis/openlog v1.1.2
        github.com/go-chassis/seclog v1.3.0
        github.com/golang/protobuf v1.4.2
        github.com/gorilla/websocket v1.4.0
        github.com/hashicorp/go-version v1.0.0
        github.com/opentracing/opentracing-go v1.1.0
        github.com/patrickmn/go-cache v2.1.0+incompatible
        github.com/prometheus/client_golang v0.9.1
        github.com/prometheus/common v0.2.0
        github.com/smartystreets/goconvey v0.0.0-20190330032615-68dc04aab96a
        github.com/stretchr/testify v1.6.1
        gopkg.in/yaml.v2 v2.3.0
        k8s.io/apimachinery v0.17.0
        k8s.io/client-go v0.17.0
)

go 1.13
```

奥卡姆剃刀法则

- 最小化容器大小
- 最小化安全漏洞
- 最小化代码量
- 最小化复杂度

依赖都去哪了？

go-chassis-extension工程




```
module github.com/go-chassis/go-chassis-extension/registry/kubernetes

go 1.13

require (
        github.com/go-chassis/go-chassis v1.8.3
        github.com/stretchr/testify v1.5.1
        k8s.io/api v0.18.3
        k8s.io/apimachinery v0.18.3
        k8s.io/client-go v0.18.3
)
```

- 按需使用扩展能力和插件

https://github.com/go-chassis/go-chassis-extension

bootstrap

```go
//InstallPlugin is a function which installs plugin,
// during initiating of go chassis, plugins will be executed
func InstallPlugin(name string, plugin Plugin) {
    bootstrapPlugins = append(bootstrapPlugins, &PluginItem{
        Name:   name,
        Plugin: plugin,
    })
}

//Bootstrap will boot plugins in orders
func Bootstrap() {
    for _, bp := range bootstrapPlugins {
        openlog.Info("Bootstrap " + bp.Name)
        if err := bp.Plugin.Init(); err != nil {
            openlog.Error(fmt.Sprintf("Failed to init #{bp.Name}. error [#{err.Error()}]"))
        }
    }
}

```

可定制的拉起过程

- 任意替换默认实现
- 加载并运行新的模块供运行时调用

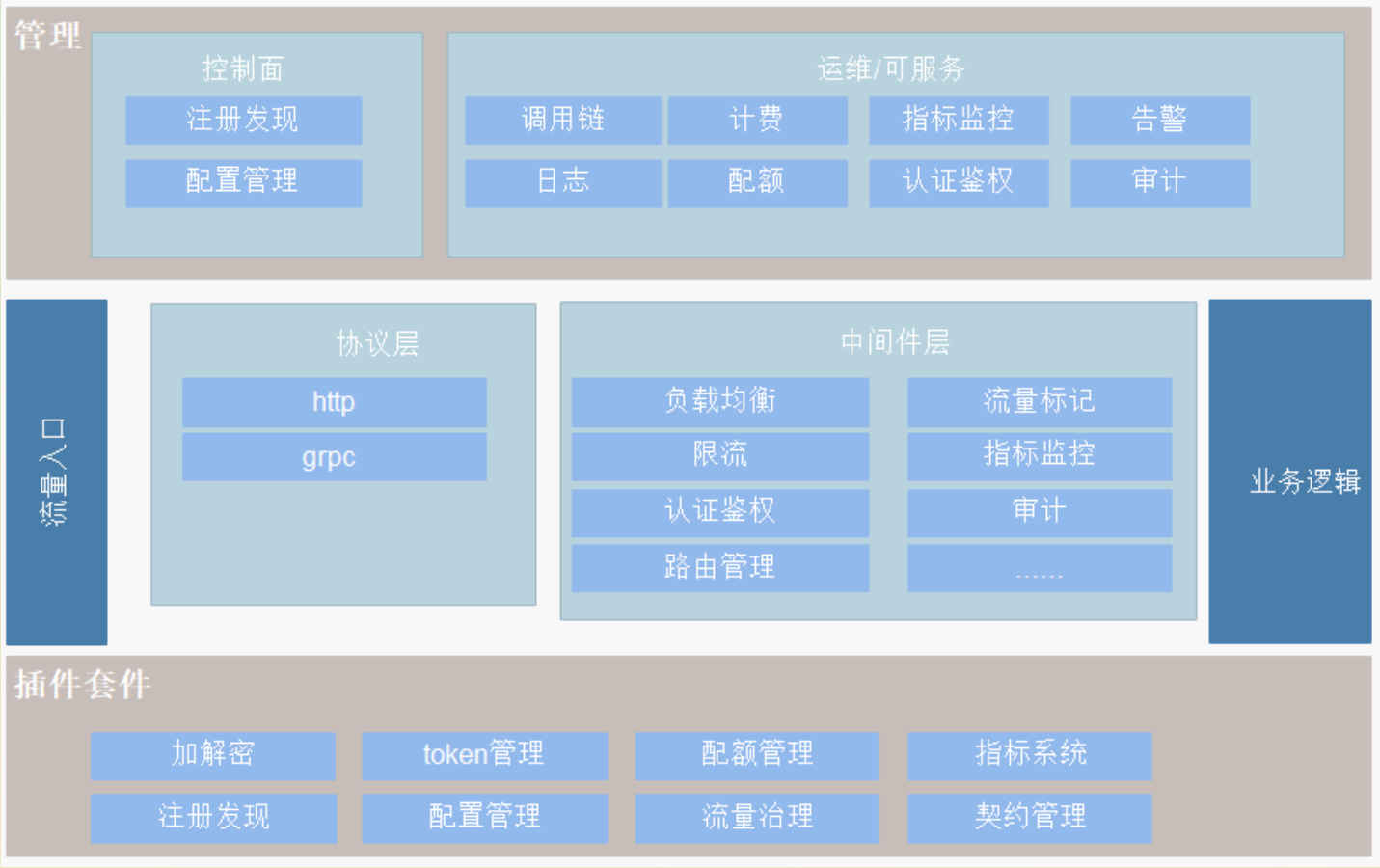
云原生生态构建

Spring cloud

- 利用开源，只做扩展，不做封装：保持原生spring cloud代码写法，并做特性增强
- 增强注册发现：提供文档管理，编写一份spring cloud 代码即可自动生成并上传到服务中心
- 路由管理：自定义规则完成金丝雀发布，蓝绿发布
- 增强配置管理：原生注解，接入到自研配置管理，提供配置一键回滚，历史管理，推送轨迹等高级特性

https://github.com/huaweicloud/spring-cloud-huawei

该不该用SideCar模式



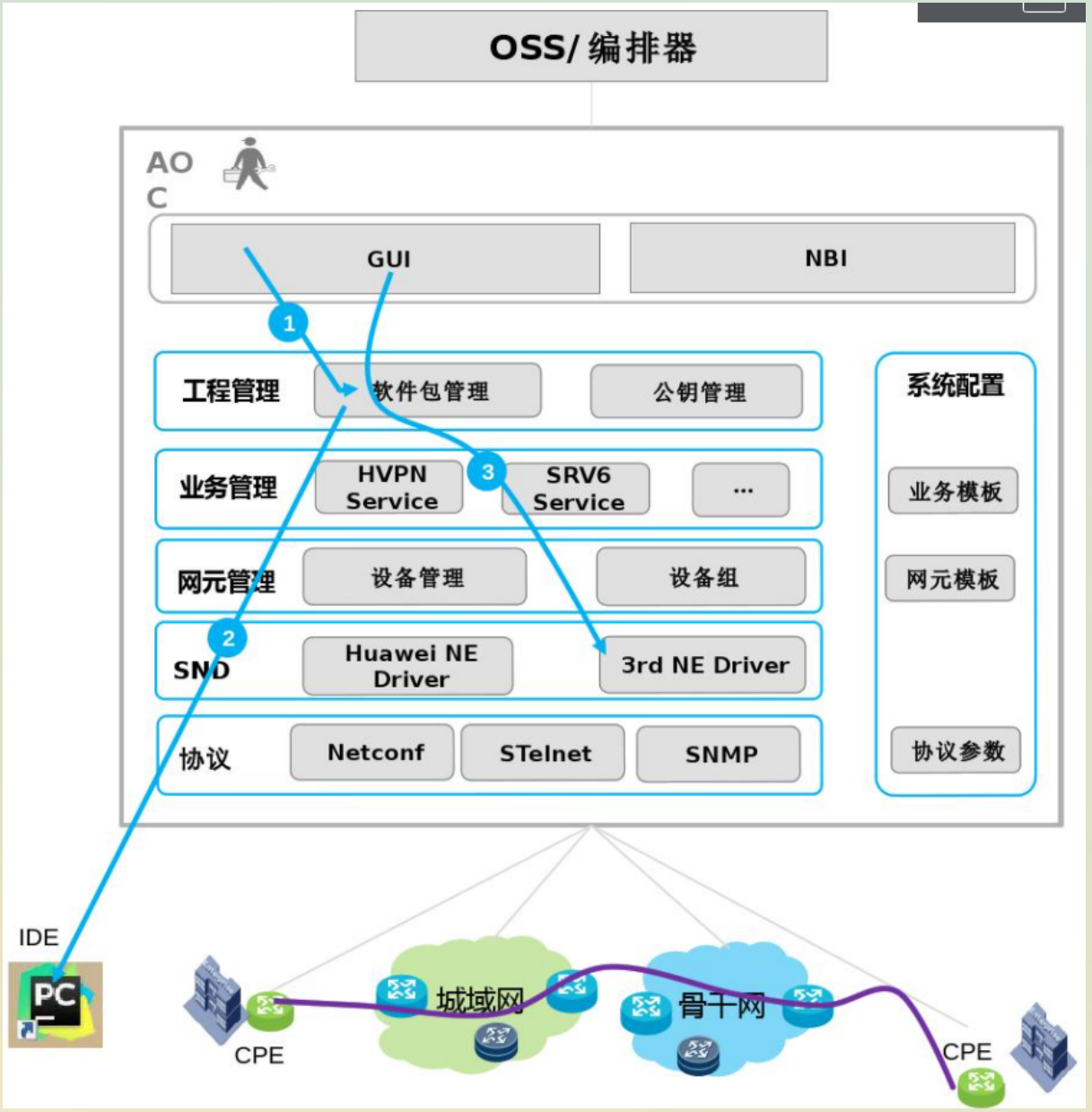
服务网格或SideCar架构模式搞定哪些部分？

- 负载均衡、通用指标监控，路由管理等能力“卸载”到服务网格，适合自己业务的才是最好的
- 卸载后，应用仍需要一个云原生开发框架，来提升团队效能，除了异构系统对接，还有业务指标，优雅停机，配置治理等能力，微服务框架可能会逐渐演进为轻量级“云应用框架”

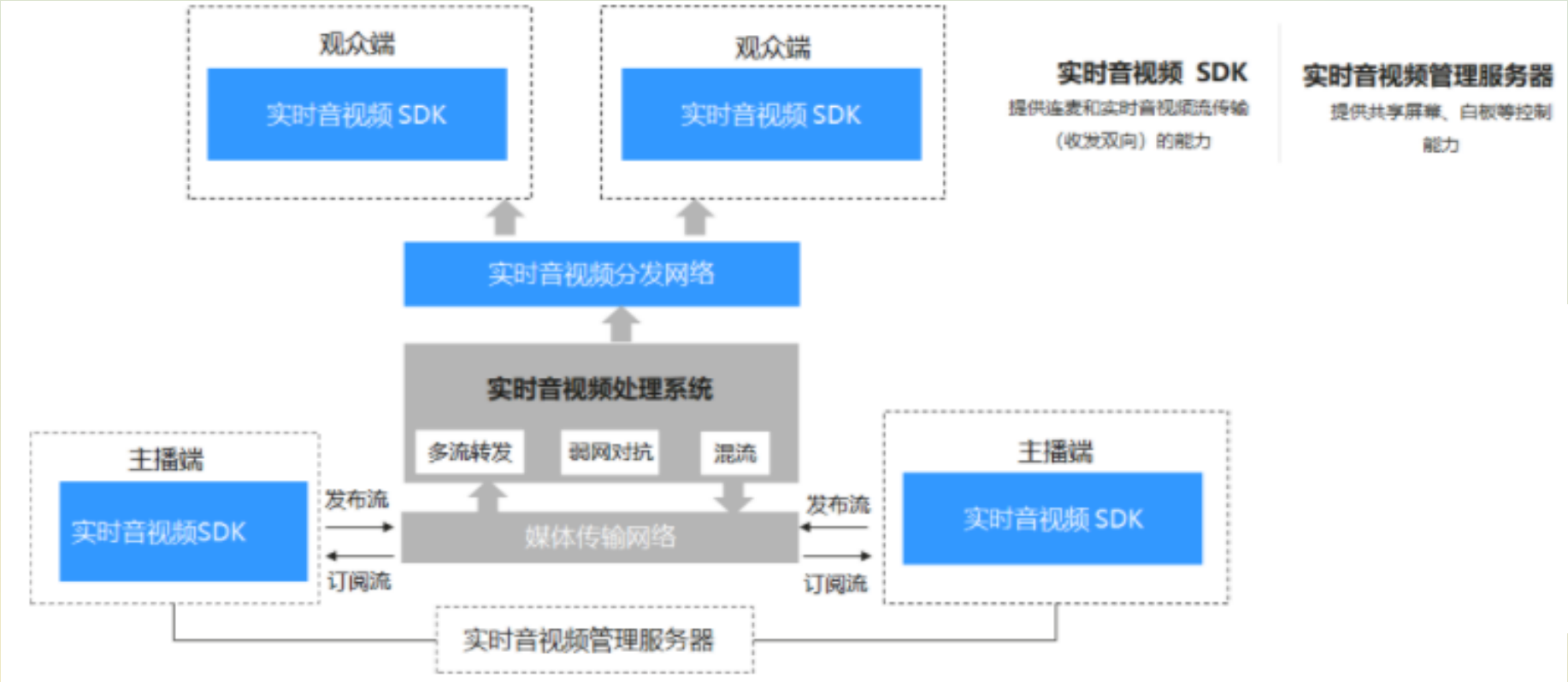
案例

华为网管产品

- 快速管理华为和三方设备
- 设备配置能力开放



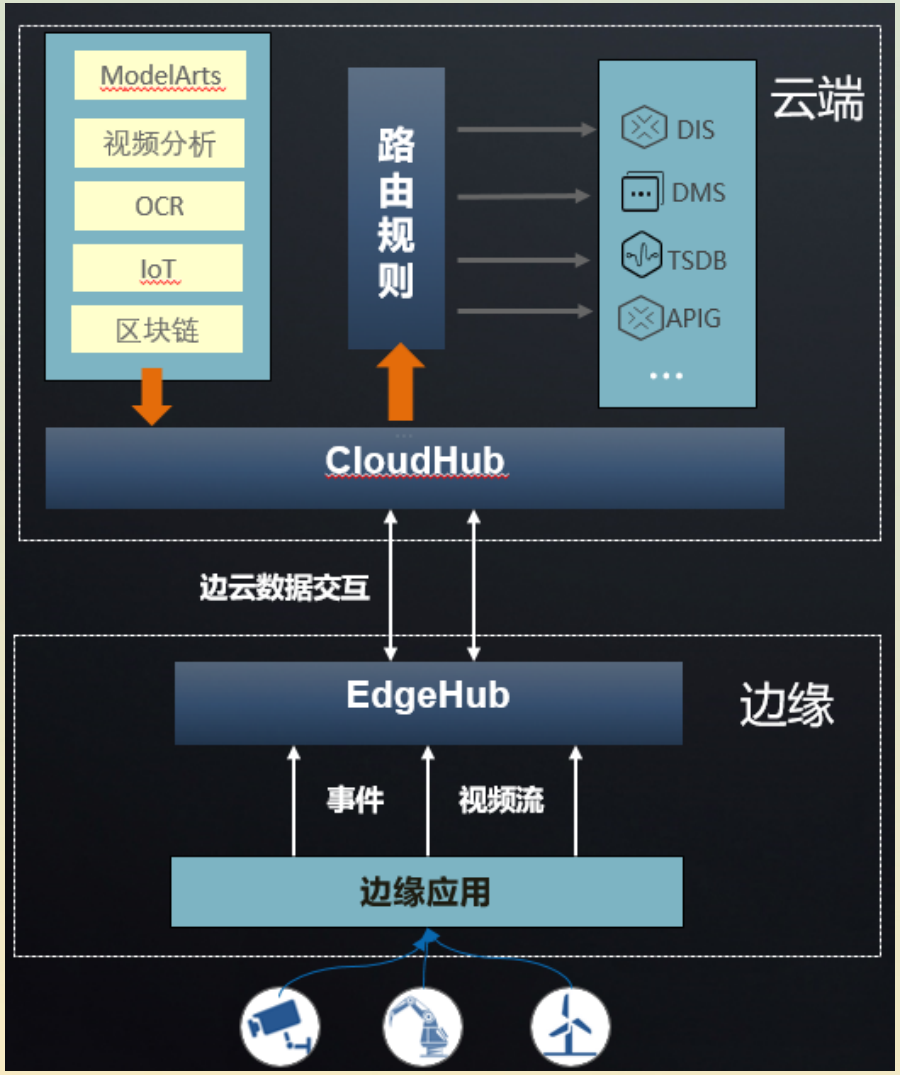
实时音视频 RTC



- 华为荣耀手机和智慧屏等终端的视频通话后台
- 已独立上线公有云，支撑终端公司畅联通话上亿注册用户，基于go-chassis进行服务治理。

产品页面：https://www.huaweicloud.com/product/rtc.html

边缘计算KubeEdge



- 基于go chassis开发服务治理底座
- 管理着全国29个省、自治区的将近10万边缘节点，超过50万边缘应用的部署。支撑了1万多个收费站中门架信息采集业务的不断调整、更新，满足了每日3亿条以上的信息采集。
- 为日后车路协同、自动驾驶等创新业务的发展提供了良好的平台支撑

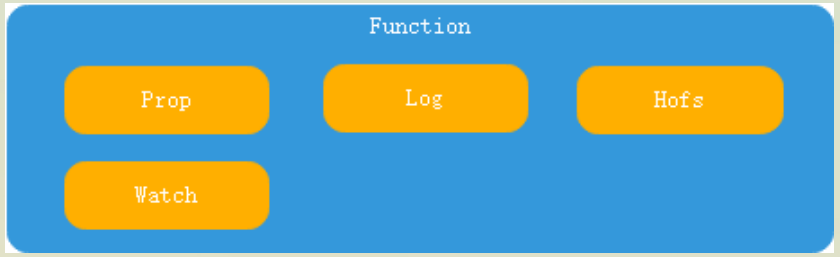
https://github.com/kubeedge/kubeedge

某框架二次封装

基于go-chassis开发。用户不直接感知服务如何启动。

用户在使用资源或业务时，直接声明即可，不用关心初始化和生命周期管理

对接系统提供的各种资源和业务，提供api接口



应用层协议支持

- 这是一种典型的案例

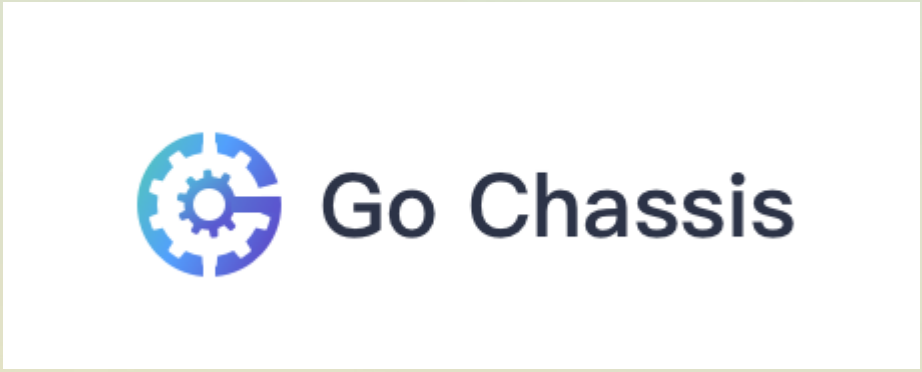
Shopee

- 分会场议题 “Go chassis在Shopee供应链的实践”

总结

1. 注册发现所承担的更多职责以及增强手段，对技术管理者反馈
2. 如何在本地进行复杂分布式系统的测试，提升研发质量
3. 实现云原生开发与治理的“通信协议”，无为而治
4. 内置服务治理、安全、可观察等多种能力，以提供稳定的云上应用， 提升研发效率

拥有自己重新制造的轮子



展望

- Go语言在新基建的机遇
- Go chassis目前的发展情况

GOPHER CHINA 2020

中国 上海 / 2020-11.21-22

Thanks

https://github.com/go-chassis/go-chassis
https://github.com/go-chassis/go-chassis-extension