# 解构 Apache/Dubbo-go
## 回顾与展望

邓明

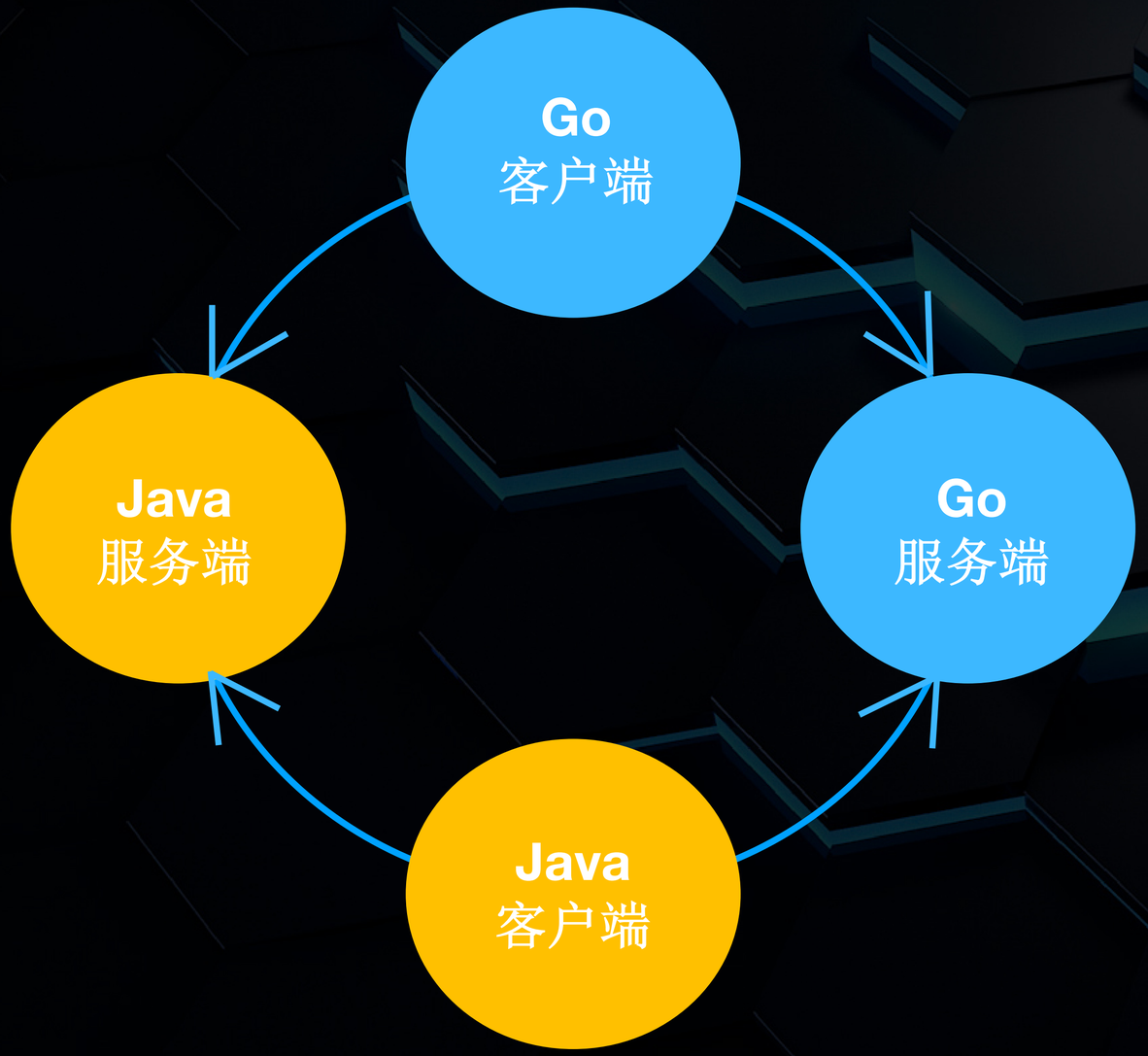
2019.12.28

# Content

- 总体框架
- Protocol——异构的基础
- 服务治理——熔断、限流
- 监控——Metrics
- 云——Kubernetes
- 展望未来

# 1. 总体框架

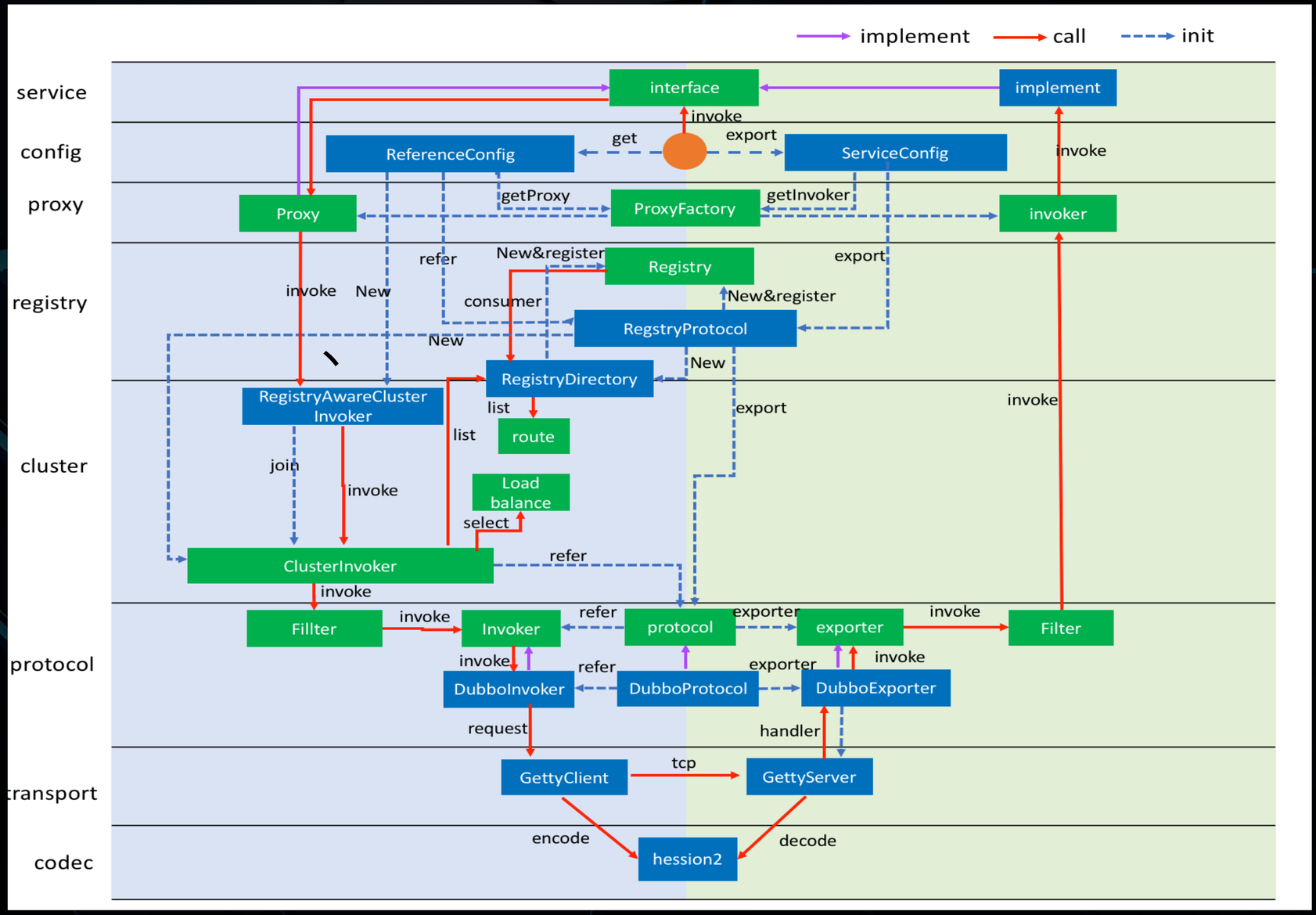
# 目标




- 同一份Go客户端代码，可调用Go服务端和Java服务端。
- 同一份Go服务端代码，可以被Go客户端和Java客户端调用。

# 整体设计

- 借鉴 dubbo 的分层设计，易拓展性
- 语言不同，部分模块实现思路不一致



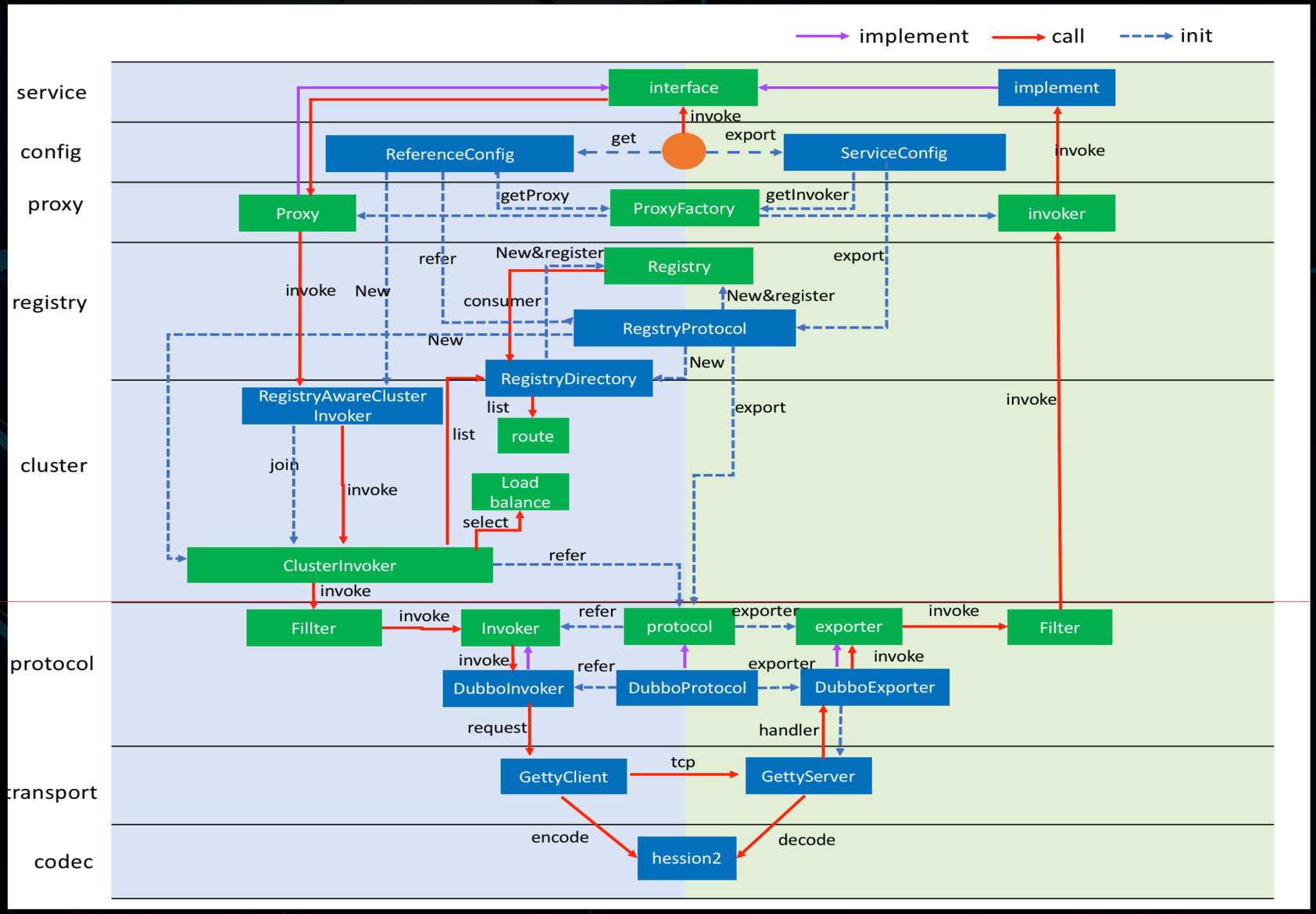
# 2. Protocol

# Protocol

## Protocol之上

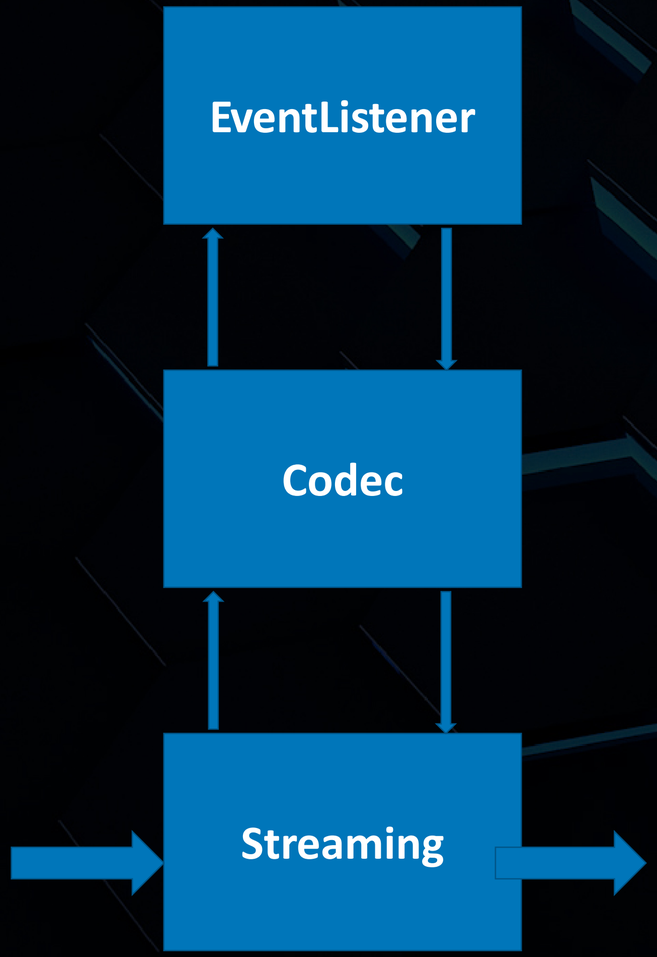
## Protocol之下

屏蔽掉RPC实现差异



# Protocol抽象与扩展

# Dubbo - Getty三层模型




## 应用层处理

处理网络包以及各种事件

- **OnOpen**
  连接过多判断、应用层连接池添加
- **OnClose/OnError**
  连接关闭或发生错误、应用层连接池清理
- **OnCron**
  定时任务、心跳(websocket除外)
- **OnMessage**
  逻辑包处理、阻塞Streaming/handleLoop

## 协议

针对不同协议分别进行序列化/反序列化

## 网络IO

网络读写的封装

# Dubbo-go-hessian2

- All JDK Exceptions
- Field Alias
- Java BigDecimal
- Java Date & Time
- Java Generic Invocation
- Dubbo Attachments
- Skipping unregistered POJO
- Emoji

# Dubbo-go-hessian2 类型映射表

| hessian type | java type | golang type |
| --- | --- | --- |
| **null** | null | nil |
| **binary** | byte[] | []byte |
| **boolean** | boolean | bool |
| **date** | java.util.Date | time.Time |
| **double** | double | float64 |
| **int** | int | int32 |
| **long** | long | int**64** |
| **string** | java.lang.String | string |
| **list** | java.util.List | slice |
| **map** | java.util.Map | map |
| **object** | custom define object | custom define struct |
| **OTHER COMMON USING TYPE** | | |
| **big decimal** | java.math.BigDecimal | github.com/dubbogo/gost/math/big/Decimal |
| **Boolean** | Boolean | *bool (TODO) |
| **Integer** | Integer | *int32(TODO) |
| **Long** | Long | *int64 (TODO) |
| **Double** | Double | *float64 (TODO) |

# 如何映射到java class

```
type POJO interface {
    JavaClassName() string
}

type POJOEnum interface {
    POJO
    String() string
    EnumValue(string) JavaEnum
}
```

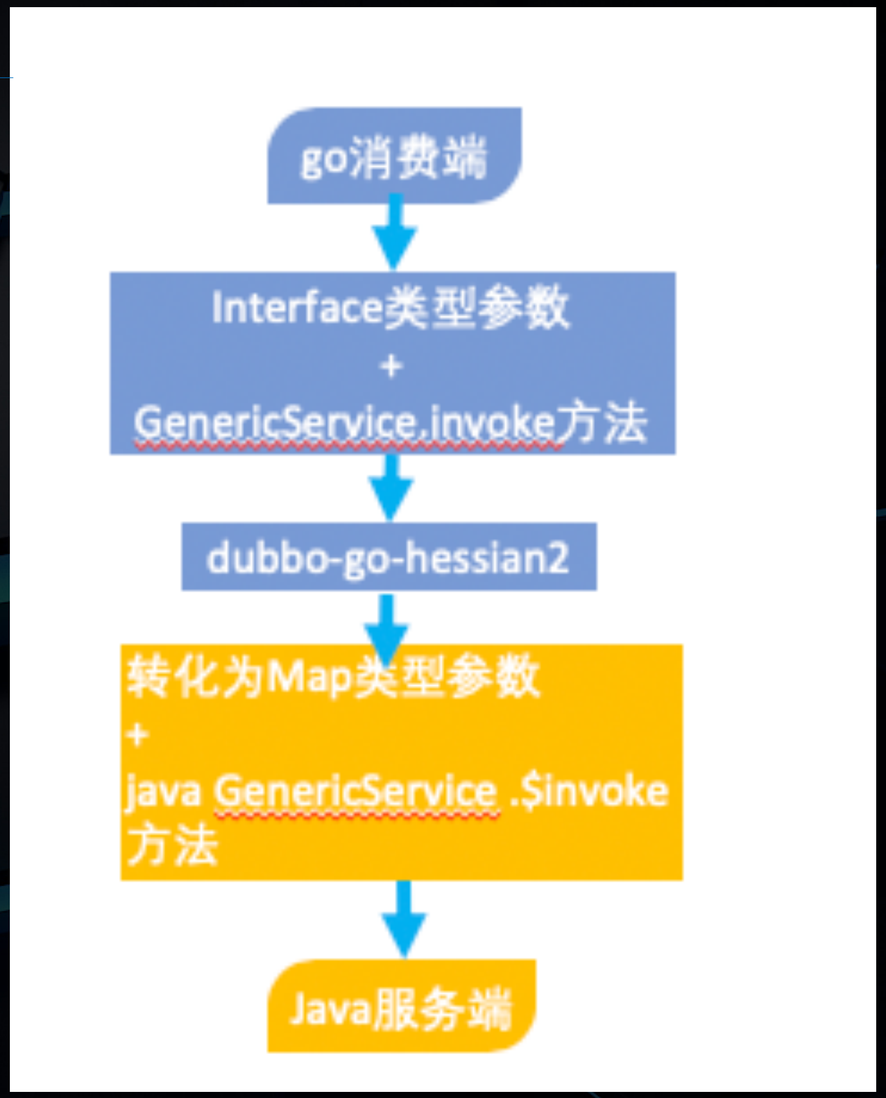
```
type User struct {
    Id    string
    Name string
    Age   int32
    Time time.Time
}
func (User) JavaClassName() string {
    return "com.ikurento.user.User"
}
```

```
type MyUser struct {
        UserFullName      string    `hessian:"user_full_name"`
        FamilyPhoneNumber string    // default convert to => familyPhoneNumber
}

func (MyUser) JavaClassName() string {
        return "com.company.myuser"
}
```

# 泛化调用

```go
type GenericService struct {
    Invoke       func(req []interface{}) (interface{}, error) `dubbo:"$invoke"`
    referenceStr string
}
```

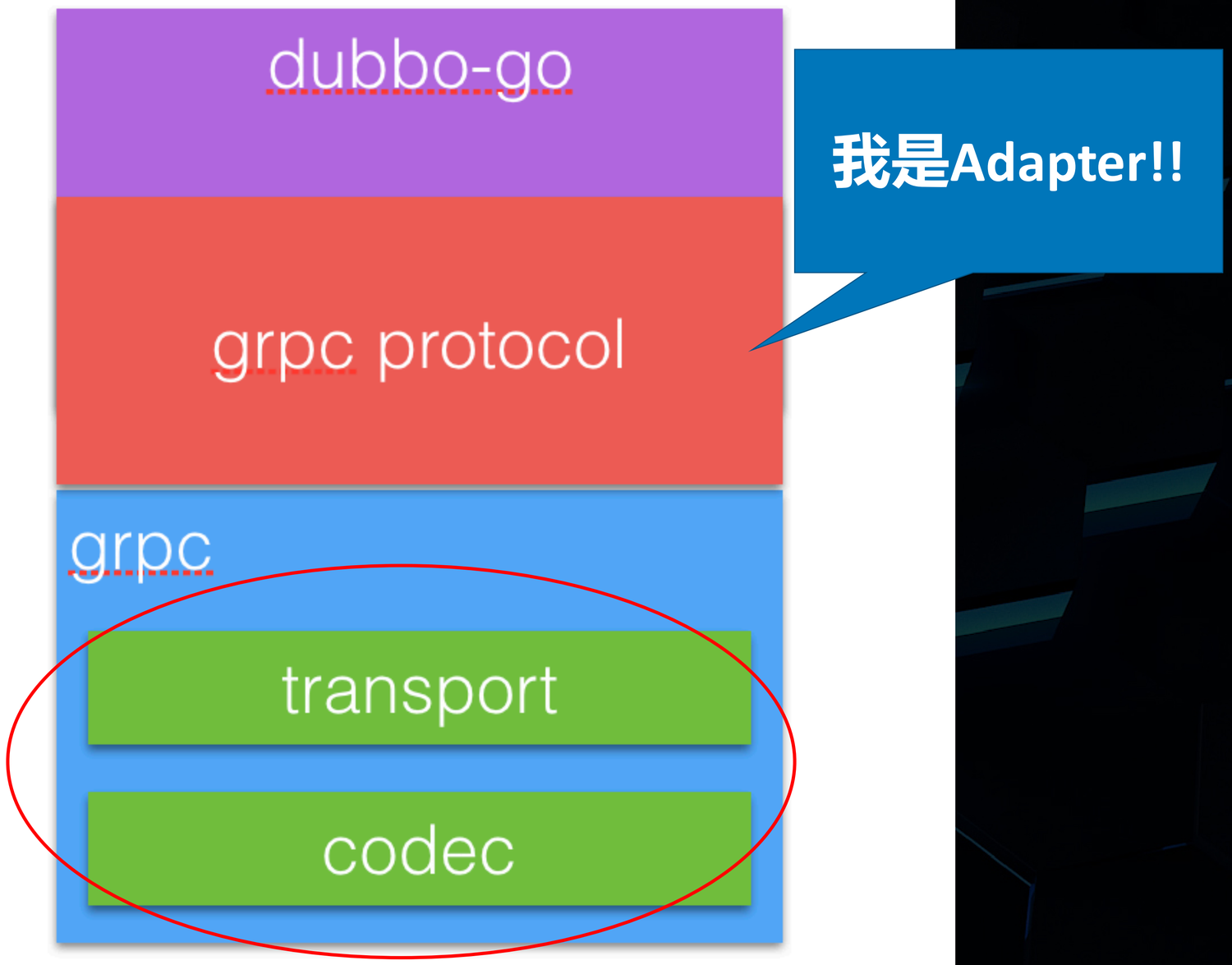
```go
referenceConfig.GenericLoad(appName) //appName is the unique identification of RPCService


time.Sleep(3 * time.Second)
println( format: "\n\n\nstart to generic invoke")
resp, err := referenceConfig.GetRPCService().(*config.GenericService).Invoke([]interface{}{"GetUser", []string{"java.lang.String"}, []interface{}{"A003"}})
```



我司已经有Java的RPC框架了，但是golang上想使用dubbogo，可以吗？

当然可以，你可以实现一个自己的Invoker，那么dubbogo的其它功能都可以使用了

# 扩展Protocol



# Grpc —— Sever端例子

```go
// server is used to implement helloworld.GreeterServer.
type server struct {
    pb.UnimplementedGreeterServer
}

// SayHello implements helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloReply, error) {
    log.Printf( format: "Received: %v", in.GetName())
    return &pb.HelloReply{Message: "Hello " + in.GetName()}, nil
}

func main() {
    lis, err := net.Listen( network: "tcp", port)
    if err != nil {
        log.Fatalf( format: "failed to listen: %v", err)
    }
    s := grpc.NewServer()
    pb.RegisterGreeterServer(s, &server{})
    if err := s.Serve(lis); err != nil {
        log.Fatalf( format: "failed to serve: %v", err)
    }
}
```

```go
175
176 func RegisterGreeterServer(s *grpc.Server, srv GreeterServer) {
177     s.RegisterService(&_Greeter_serviceDesc, srv)
178 }
179
```

```
    key := url.GetParam(constant.BEAN_NAME_KEY, d: "")
    service := config.GetProviderService(key)

    ds, ok := service.(DubboGrpcService)
    if !ok {
        panic( v: "illegal service type registered")
    }

    m, ok := reflect.TypeOf(service).MethodByName("SetProxyImpl")
    if !ok {
        panic( v: "method SetProxyImpl is necessary for grpc service")
    }

    exporter, _ := grpcProtocol.ExporterMap().Load(url.ServiceKey())
    if exporter == nil {
        panic(fmt.Sprintf( format: "no exporter found for servicekey: %v", url.ServiceKey()))
    }
    invoker := exporter.(protocol.Exporter).GetInvoker()
    if invoker == nil {
        panic(fmt.Sprintf( format: "no invoker found for servicekey: %v", url.ServiceKey()))
    }
    in := []reflect.Value{reflect.ValueOf(service)}
    in = append(in, reflect.ValueOf(invoker))
    m.Func.Call(in)

    server.RegisterService(ds.ServiceDesc(), service)

    s.grpcServer = server
    if err = server.Serve(lis); err != nil {
        panic(err)
    }
}
```

# Grpc —— client端例子

```go
func main() {
    // Set up a connection to the server.
    conn, err := grpc.Dial(address, grpc.WithInsecure(), grpc.WithBlock())
    if err != nil {
        log.Fatalf( format: "did not connect: %v", err)
    }
    defer conn.Close()
    c := pb.NewGreeterClient(conn)

    // Contact the server and print out its response.
    name := defaultName
    if len(os.Args) > 1 {
        name = os.Args[1]
    }
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    r, err := c.SayHello(ctx, &pb.HelloRequest{Name: name})
    if err != nil {
        log.Fatalf( format: "could not greet: %v", err)
    }
    log.Printf( format: "Greeting: %s", r.GetMessage())
}
```

```go
type greeterClient struct {
    cc *grpc.ClientConn
}

func NewGreeterClient(cc *grpc.ClientConn) GreeterClient {
    return &greeterClient{cc}
}

func (c *greeterClient) SayHello(ctx context.Context, in *HelloRequest, opts ...grpc.CallOption) (*HelloReply, error) {
    out := new(HelloReply)
    err := c.cc.Invoke(ctx, "/helloworld.Greeter/SayHello", in, out, opts...)
    if err != nil {
        return nil, err
    }
    return out, nil
}
```

```go
type Client struct {
    *grpc.ClientConn
    invoker reflect.Value
}

func NewClient(impl interface{}, url common.URL) *Client {

    conn, err := grpc.Dial(url.Location, grpc.WithInsecure(), grpc.WithBlock())
    if err != nil {
        panic(err)
    }

    in := []reflect.Value{}
    in = append(in, reflect.ValueOf(conn))
    method := reflect.ValueOf(impl).MethodByName( name: "GetDubboStub")
    res := method.Call(in)
    invoker := res[0].Interface()

    return &Client{
        ClientConn: conn,
        invoker:    reflect.ValueOf(invoker),
    }
}
```

```go
func (gi *GrpcInvoker) Invoke(invocation protocol.Invocation) protocol.Result {
    var (
        result protocol.RPCResult
    )

    if invocation.Reply() == nil {
        result.Err = ErrNoReply
    }

    in := []reflect.Value{}
    in = append(in, reflect.ValueOf(context.Background()))
    in = append(in, invocation.ParameterValues()...)

    methodName := invocation.MethodName()
    method := gi.client.invoker.MethodByName(methodName)
    res := method.Call(in)

    result.Rest = res[0]
    // check err
    if !res[1].IsNil() {
        result.Err = res[1].Interface().(error)
    } else {
        _ = hessian2.ReflectResponse(res[0], invocation.Reply())
    }

    return &result
}
```

# 3. 服务治理

# 单机熔断

## 判断参数

MaxConcurrentRequests、Timeout、ErrorPercentThreshold

## 保护窗口

保护窗口时长

## 保护动作

user command

# 单机限流

## 限流对象

Interface? Service? Method? IP? UserID? fellow/guest ?
CPU/thread/memory?

## 限流方法

单机/集群qps？自适应限流？分级限流？

## 限流动作

限流判定后怎么处理？

# 单机限流——dubbo接口

```
 * </pre>
 */
public interface TPSLimiter {

    /**
     * judge if the current invocation is allowed by TPS rule
     *
     * @param url        url
     * @param invocation invocation
     * @return true allow the current invocation, otherwise, return false
     */
    boolean isAllowable(URL url, Invocation invocation);

}
```

# 单机限流——dubbo实现

```java
@Activate(group = CommonConstants.PROVIDER, value = TPS_LIMIT_RATE_KEY)
public class TpsLimitFilter implements Filter {

    private final TPSLimiter tpsLimiter = new DefaultTPSLimiter();

    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {

        if (!tpsLimiter.isAllowable(invoker.getUrl(), invocation)) {
            throw new RpcException(
                    "Failed to invoke service " +
                            invoker.getInterface().getName() +
                            "." +
                            invocation.getMethodName() +
                            " because exceed max service tps.");
        }

        return invoker.invoke(invocation);
    }

}
```

# 单机限流——dubbo-go接口

```go
type TpsLimiter interface {
    IsAllowable(common.URL, protocol.Invocation) bool
}
```

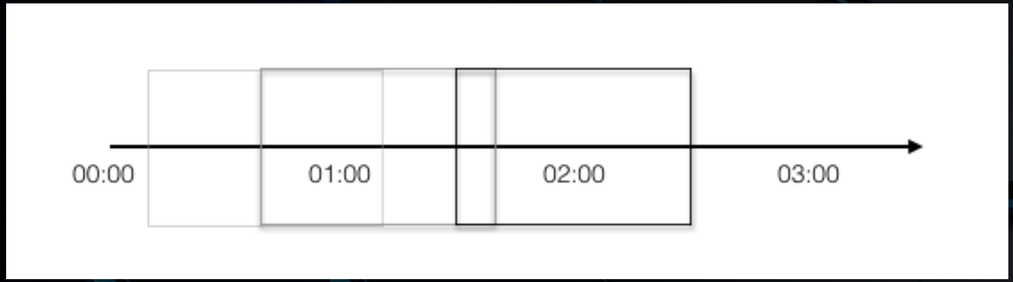
```go
type TpsLimitStrategy interface {
    IsAllowable() bool
}
```

```go
type RejectedExecutionHandler interface {
    RejectedExecution(url common.URL, invocation protocol.Invocation) protocol.Result
}
```

# 单机限流——固定窗口


```
63
64  func (impl *FixedWindowTpsLimitStrategyImpl) IsAllowable() bool {
65
66      current := time.Now().UnixNano()
67      if impl.timestamp+impl.interval < current {
68          // it's a new window
69          // if a lot of threads come here, the count will be set to 0 several times.
70          // so the return statement will be wrong.
71          impl.timestamp = current
72          impl.count = 0
73      }
74      // this operation is thread-safe, but count + 1 may be overflow
75      return atomic.AddInt32(&impl.count, delta: 1) <= impl.rate
76  }
77
```

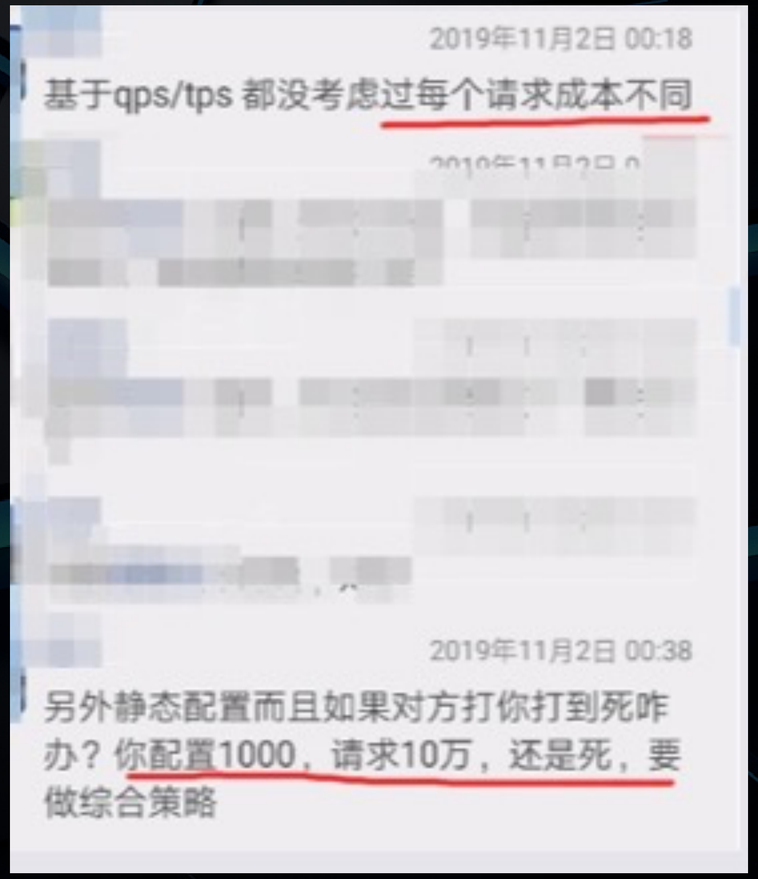
# 单机限流——滑动窗口



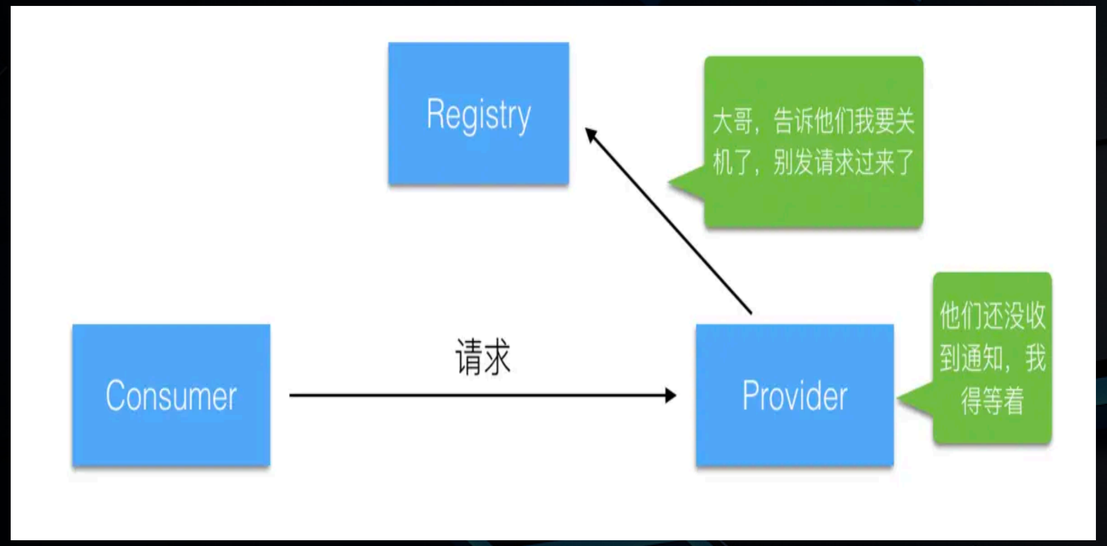

```go
func (impl *SlidingWindowTpsLimitStrategyImpl) IsAllowable() bool {
    impl.mutex.Lock()
    defer impl.mutex.Unlock()
    // quick path
    size := impl.queue.Len()
    current := time.Now().UnixNano()
    if size < impl.rate {
        impl.queue.PushBack(current)
        return true
    }

    // slow path
    boundary := current - impl.interval

    timestamp := impl.queue.Front()
    // remove the element that out of the window
    for timestamp != nil && timestamp.Value.(int64) < boundary {
        impl.queue.Remove(timestamp)
        timestamp = impl.queue.Front()
    }
    if impl.queue.Len() < impl.rate {
        impl.queue.PushBack(current)
        return true
    }
    return false
}
```

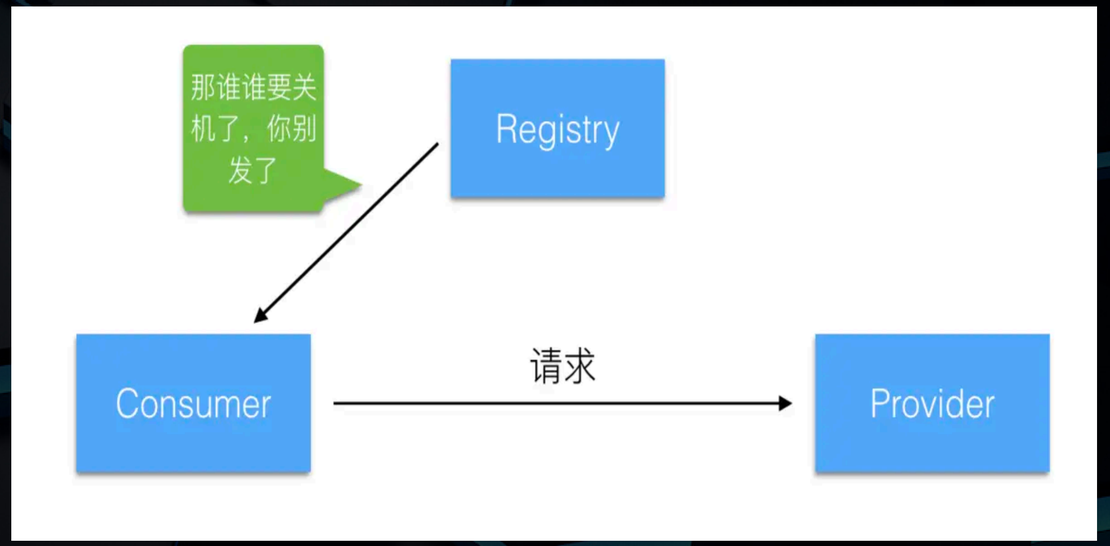
# 单机限流——待改进

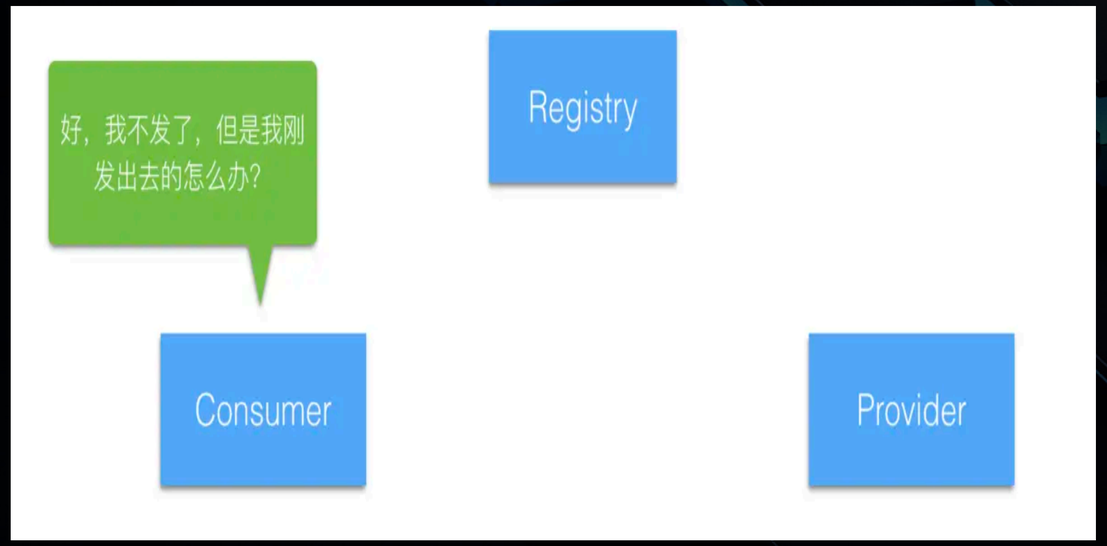

1000 requests, accepted
1000 requests, rejected
繁忙
空闲
00:00
00:10
01:00

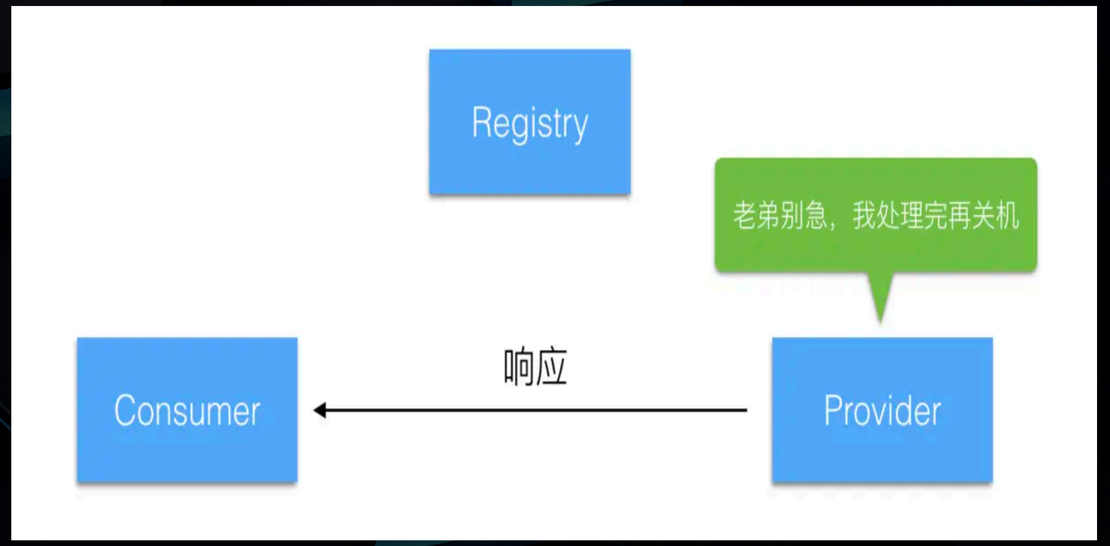

2019年11月2日 00:18
基于qps/tps 都没考虑过每个请求成本不同
2019年11月2日 00:38
另外静态配置而且如果对方打你打到死咋办？你配置1000，请求10万，还是死，要做综合策略

# 优雅退出

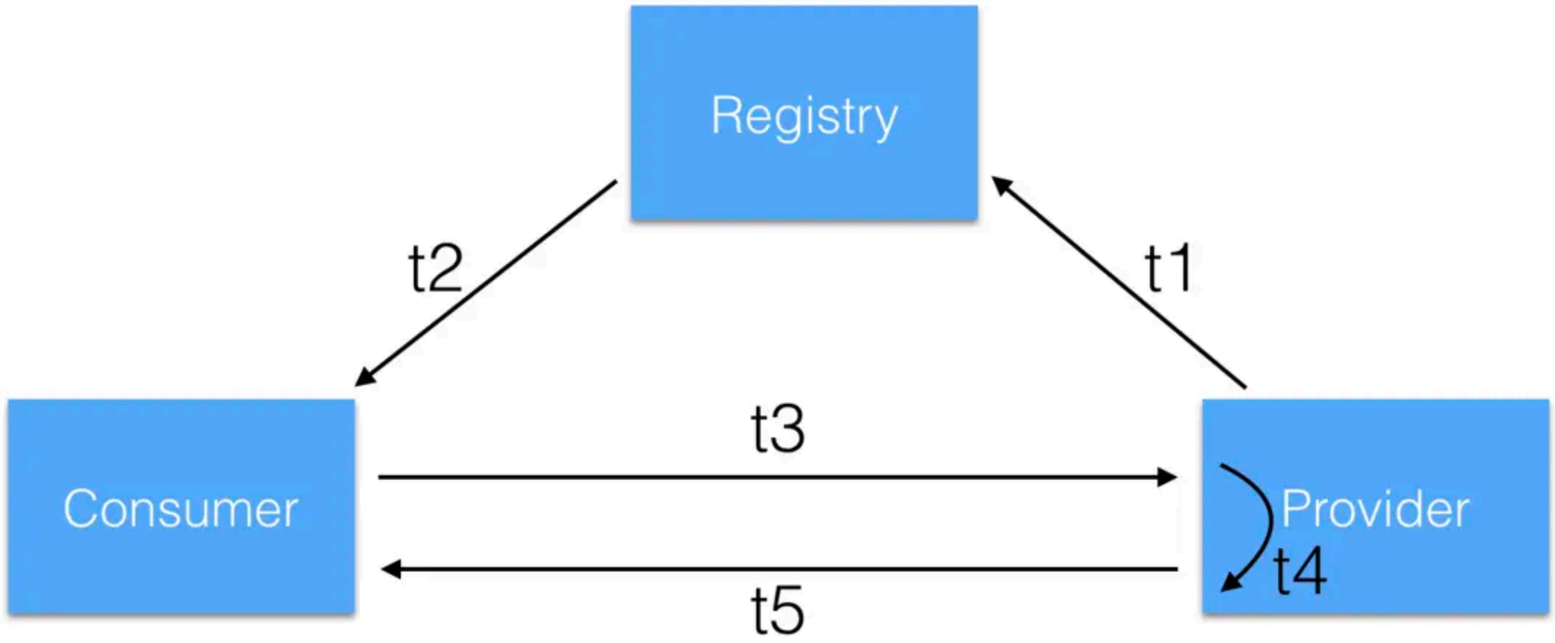

Registry
大哥，告诉他们我要关机了，别发请求过来了
Consumer
请求
Provider
他们还没收到通知，我得等着


那谁谁要关机了，你别发了
Registry
Consumer
请求
Provider


好，我不发了，但是我刚发出去的怎么办?
Registry
Consumer
Provider


Registry
老弟别急，我处理完再关机
响应
Consumer
Provider

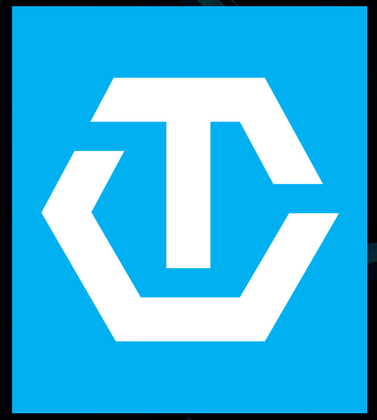
# 优雅退出——等待时长




step_timeout>t1+t2+t3+t4+t5

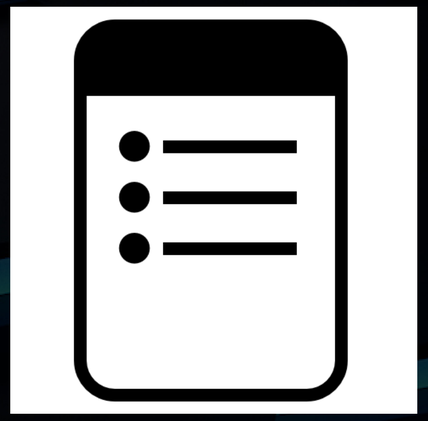
# 4. 监控

# 监控类型



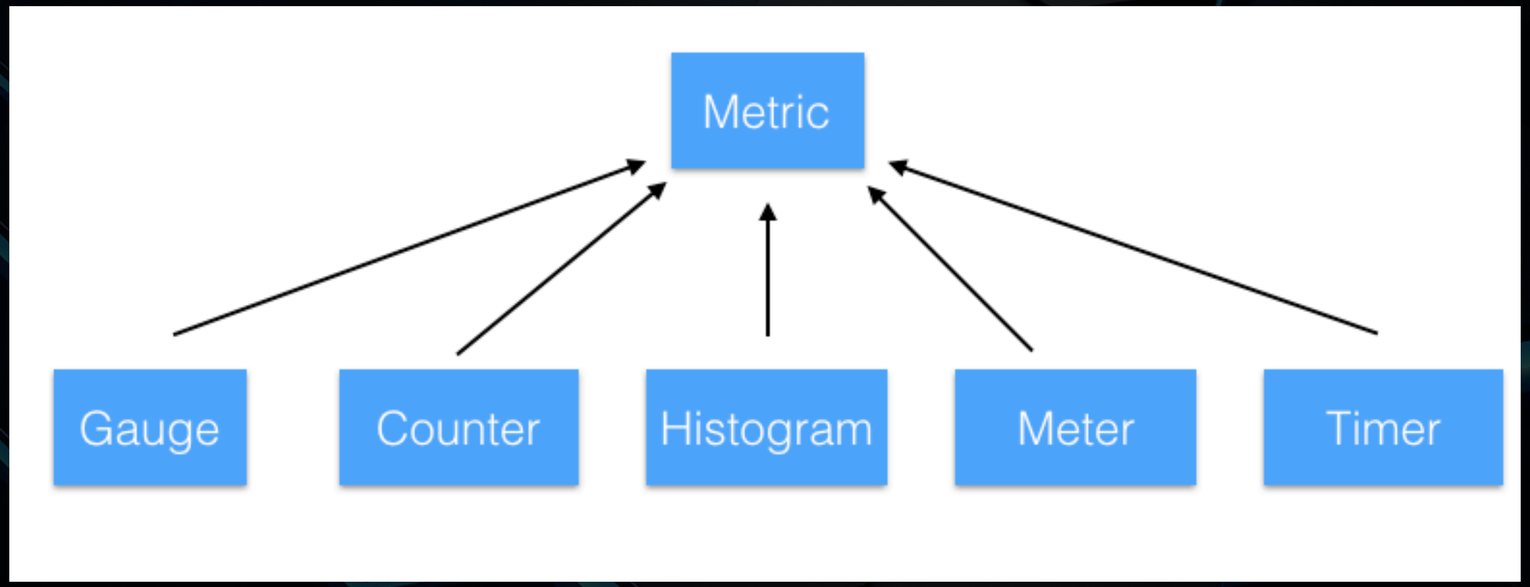
Tracing

Metrics



Logging

# Metrics



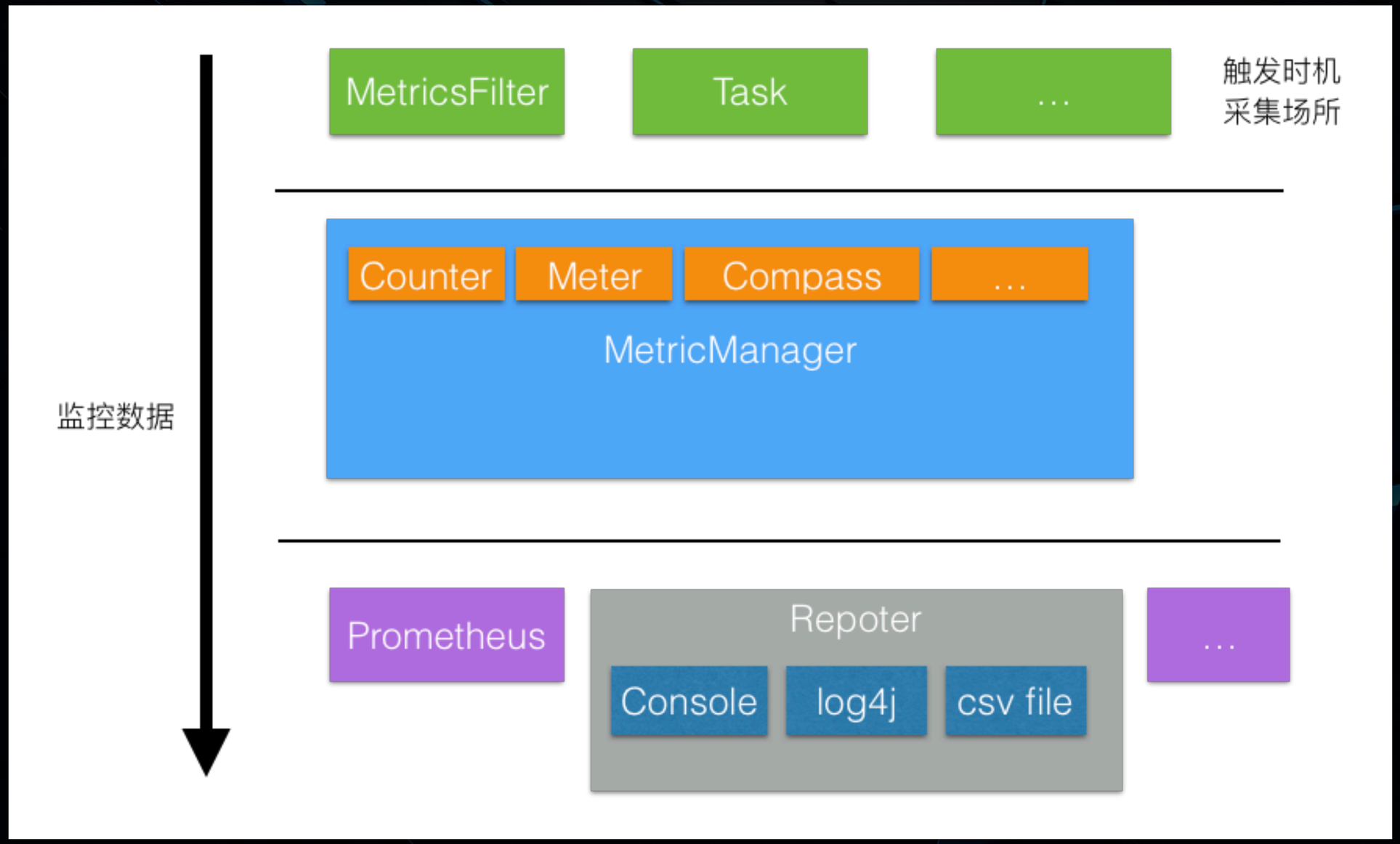

Gauge: 一种实时数据的度量，反映的是瞬态的数据，不具有累加性；

Counter: 计数器型指标，适用于记录调用总量等类型的数据；

Histogram: 直方分布指标，例如，可以用于统计某个接口的响应时间，可以展示50%, 70%, 90%的请求响应时间落在哪个区间内；

Meter: 一种用于度量一段时间内吞吐率的计量器。例如，一分钟内，五分钟内，十五分钟内的qps指标；

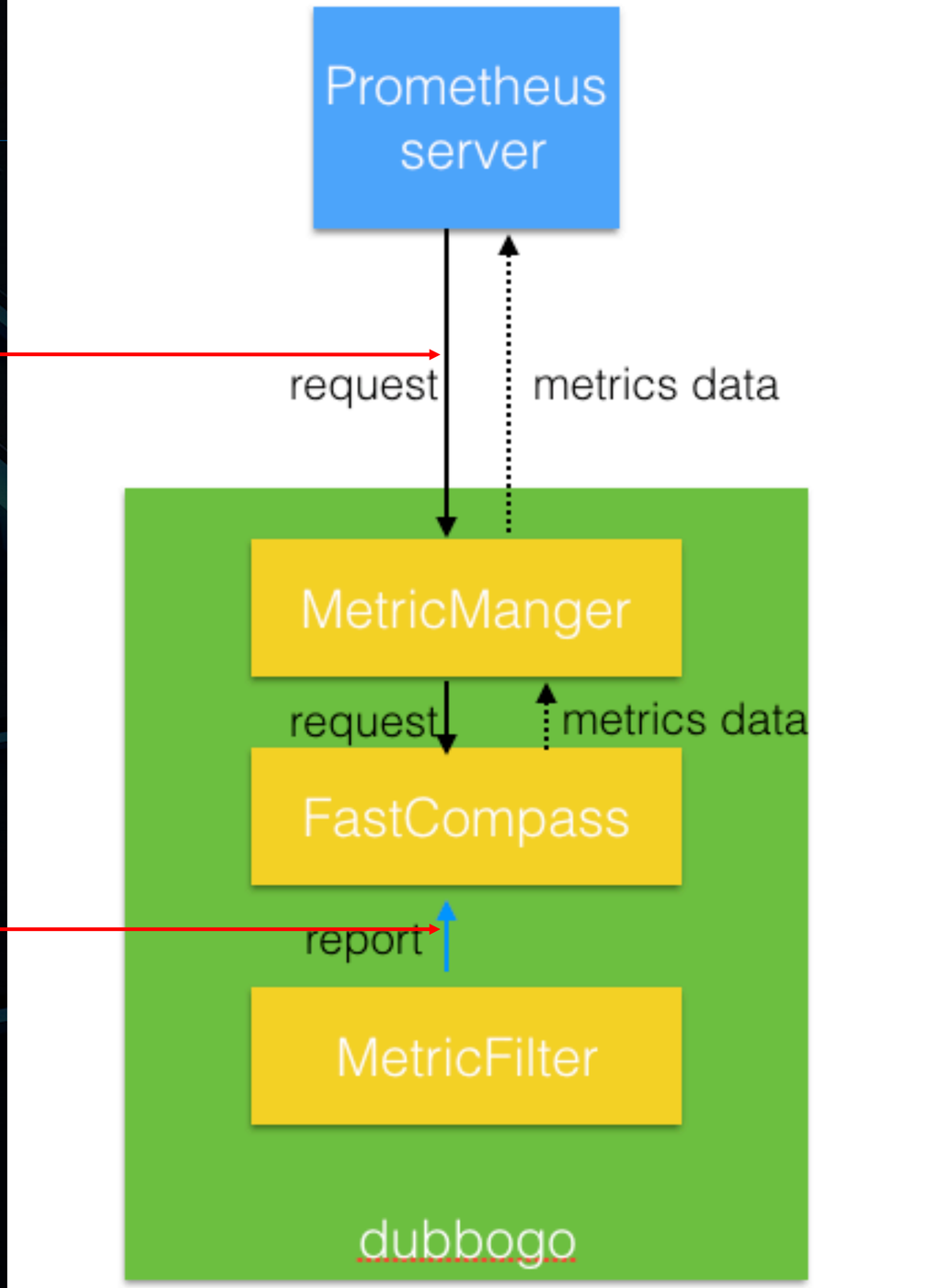
# Metrics


MetricsFilter
Task
...
触发时机
采集场所
Counter
Meter
Compass
...
MetricManager
监控数据
Prometheus
Repoter
Console
log4j
csv file
...

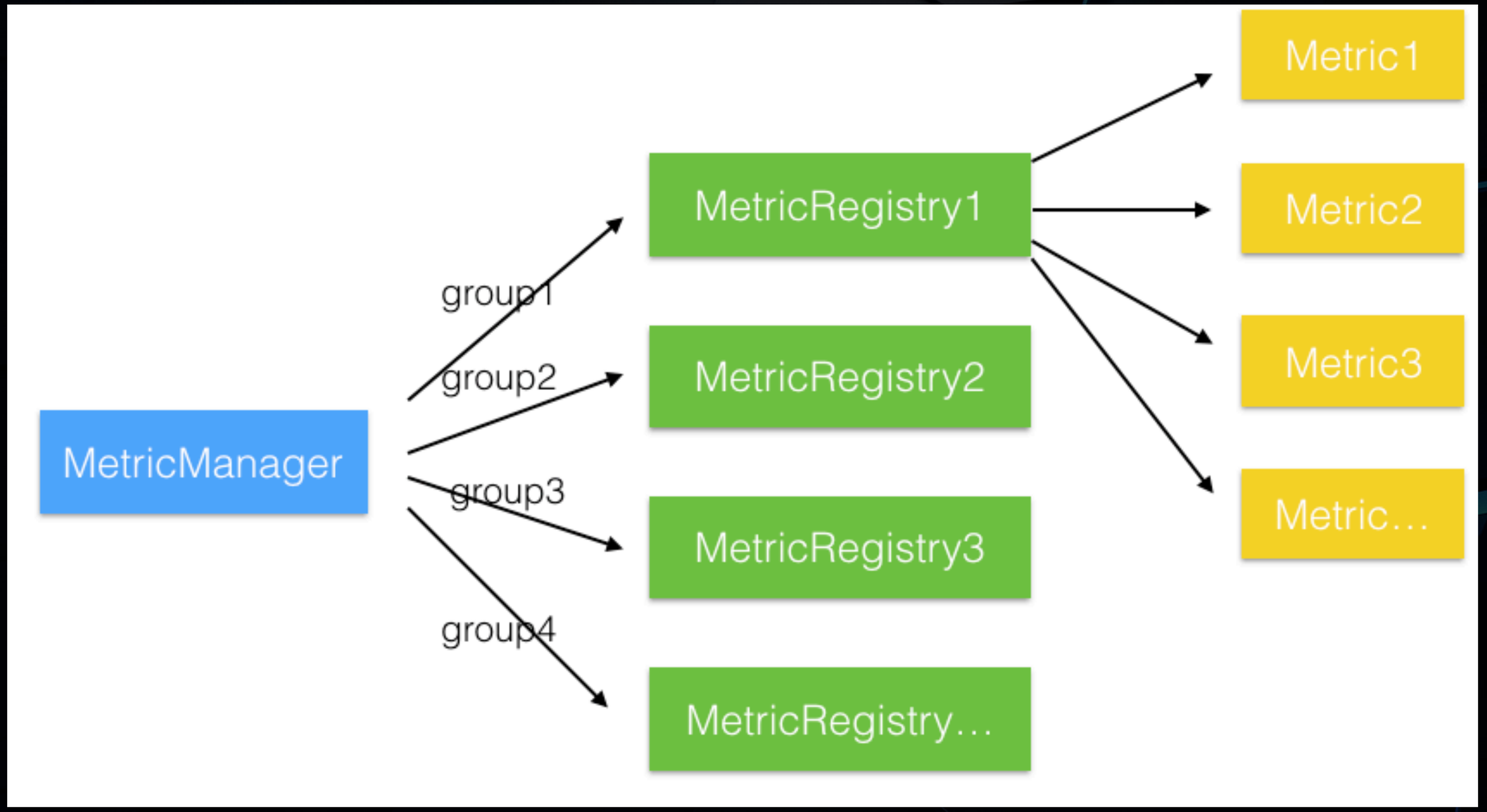
# Metrics

定时请求

实时采集



# Metrics


MetricManager
group1
group2
group3
group4
MetricRegistry1
MetricRegistry2
MetricRegistry3
MetricRegistry…
Metric1
Metric2
Metric3
Metric…

# 5. Kubernetes

# Kubernetes VS dubbo-go

k8s service：一套完全独立的服务管理体系

dubbo interface：

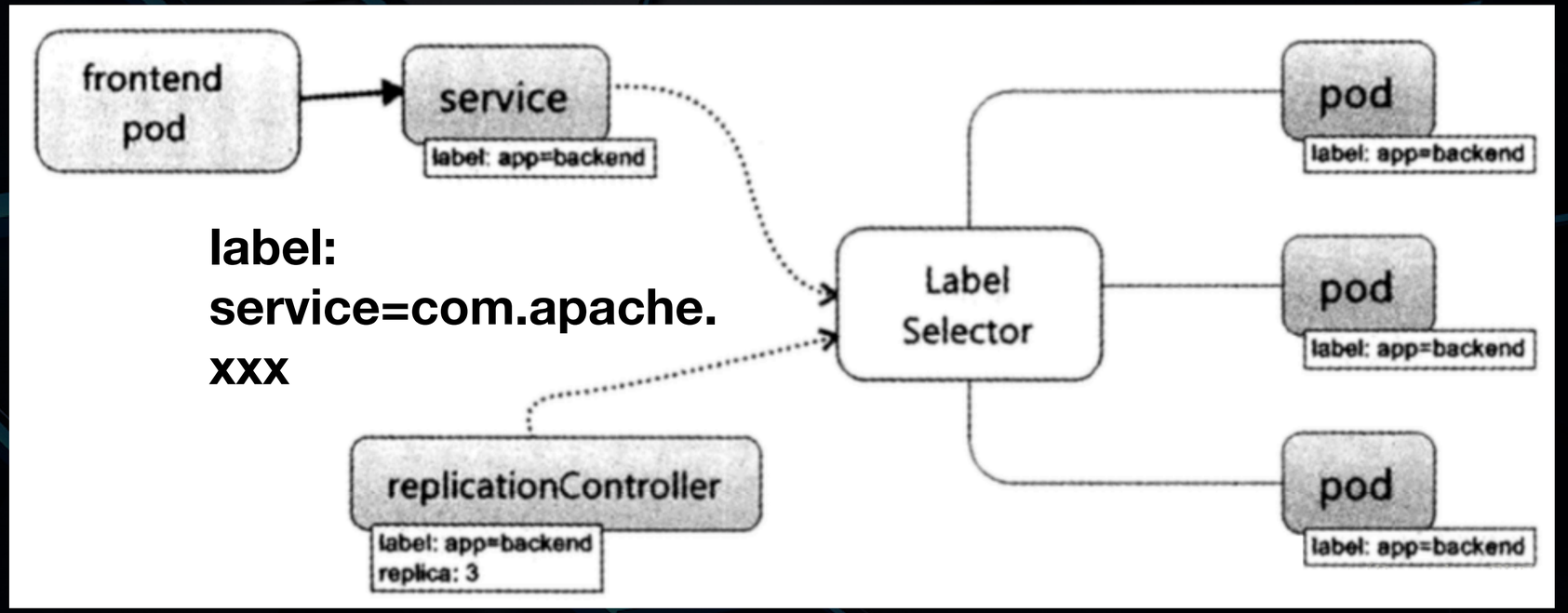
key: interface/version/group
value: ip:port/revision/ service/method/role

k8s service ≈ dubbo Directory + Router + Load Balance + Filter

# Kubernetes注册中心——Service维度

- dubbo的interface接口，放到Service 的annotations中作为服务发现的信息
- 通过label selector将符合label选出作为这个service的endpoint

缺点：无法监听、使用k8s的健康检查做服务发现

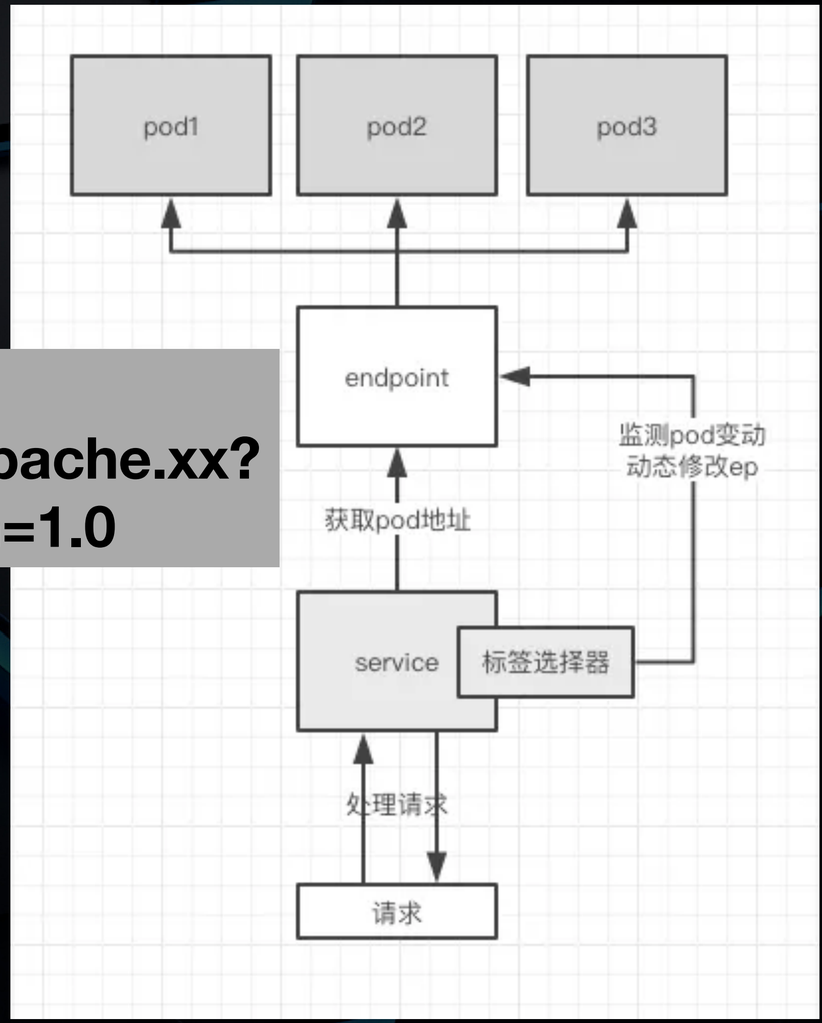


# Kubernetes注册中心——endpoint维度

- 服务注册：将每个**pod**提供的**interface**信息放到**endpoint**的**metadata**的**annotation**中注册
- 监听**k8s** 的**api server**中的对应的**endpoint**的变化，利用**k8s**健康检查，有效剔除无用的节点

缺点：一个**endpoint**代表一个服务发现的监控单元，需要根据每个**service**的**group**和**version**组合，生成多个**endpoint**

**Metadata：annotation:com.apache.xx?group=xx&version=1.0**



# Kubernetes注册中心——Operator维度

目前k8s 提供operator的方式来发布应用，operator可以定制化 。

**dubbo-go operator:**

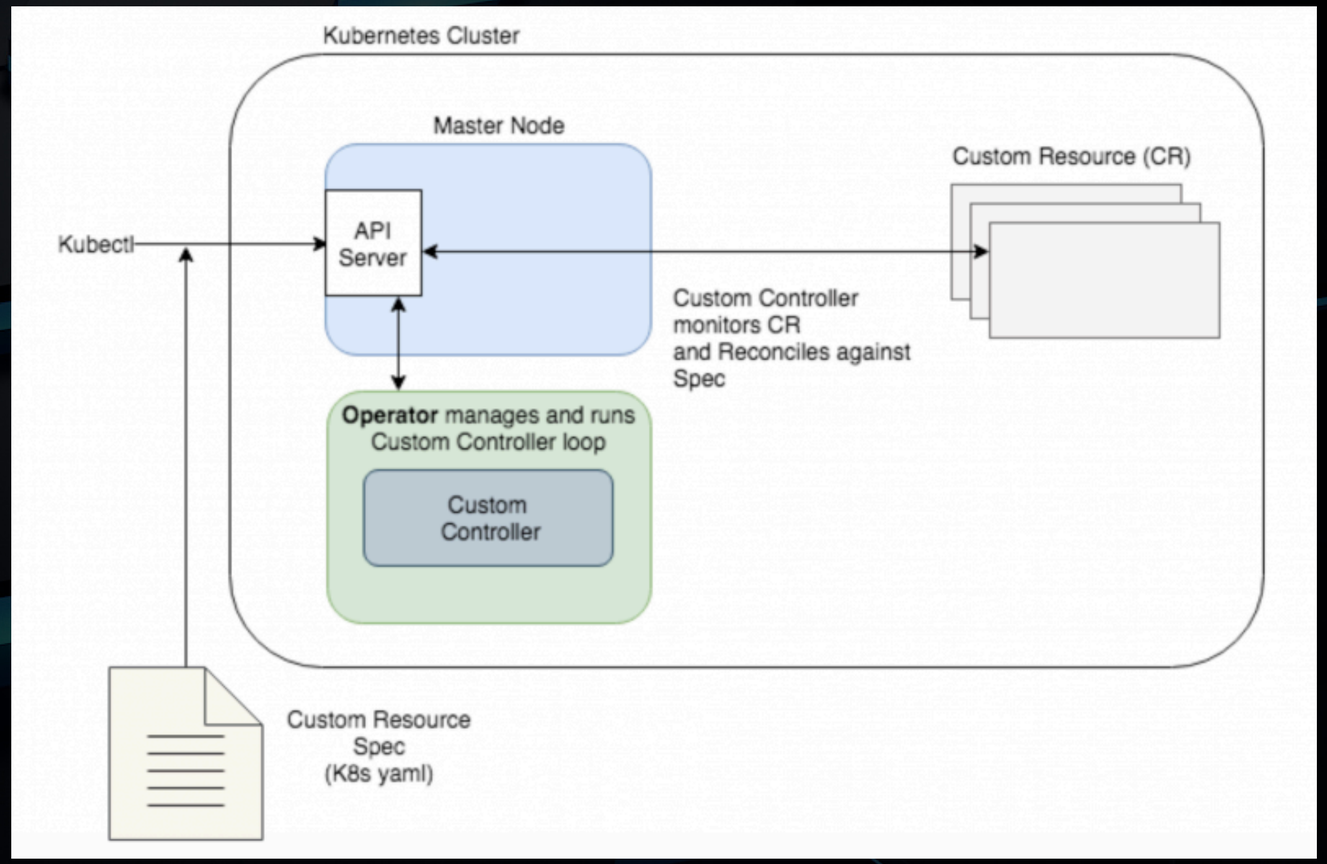
- 支持服务注册，通过k8s api server 以crd的形式写入k8s etcd
- 支持服务的健康检查，及时的剔除不健康的服务

**dubbo-go:**

- 通过k8s api server 对crd资源进行查询
- crd 资源的监听，及时变更crd 资源的变化

优点：主流方案，高度定制化

挑战：单独维护operator的运维成本



# 5. 展望未来

# 未来可期

1 Dubbo Router

2 Tracing & Metrics & Devops

3 Kubernetes Operator

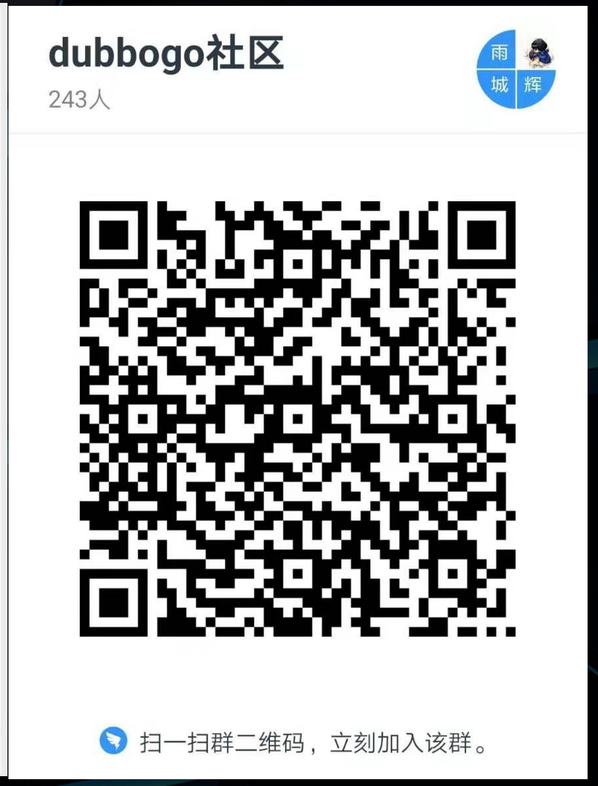
4 Cloud Native

# 社区介绍

apache / dubbo-go　Unwatch 110　Unstar 1.3k　Fork 203

apache / dubbo-go-hessian2　Unwatch 32　Unstar 63　Fork 44

- 记录在issuse上的使用单位有12个
- pr数量191个
- issue数量68个

# 落地实践

# Thank you !