

Go微服务实战

——毛剑

Agenda

✧ 微服务的演进

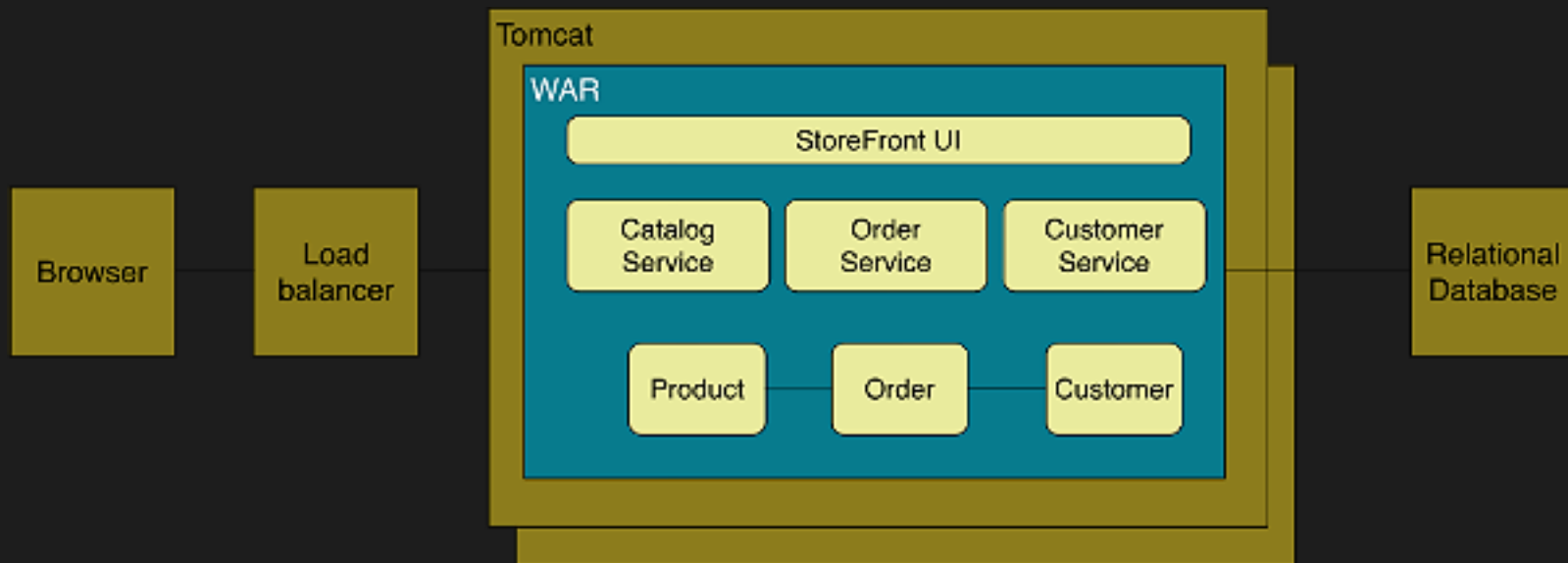
✧ 高可用

✧ 中间件

✧ 持续集成和交付

✧ 运维体系

微服务的演进



微服务的演进

✧梳理业务边界

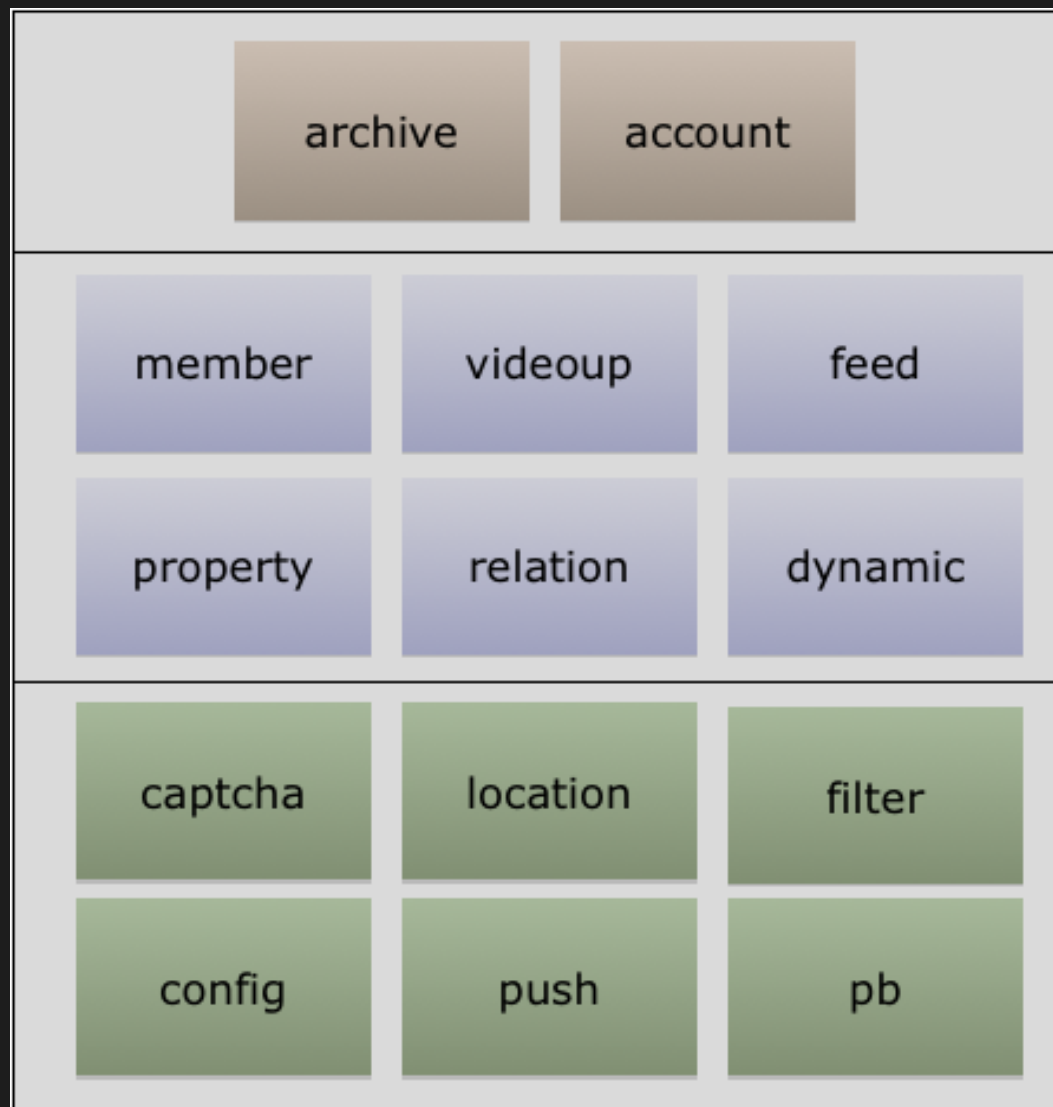
✧资源隔离部署

✧内外网服务隔离

✧RPC框架

✧API Gateway

微服务的演进



微服务的演进

✧ 梳理业务边界

✧ 资源隔离部署

✧ 内外网服务隔离

✧ RPC框架

✧ API Gateway

微服务的演进



昵图网 www.nipic.com

By:yu653310207 No.20120907114106411106

微服务的演进

✧ 梳理业务边界

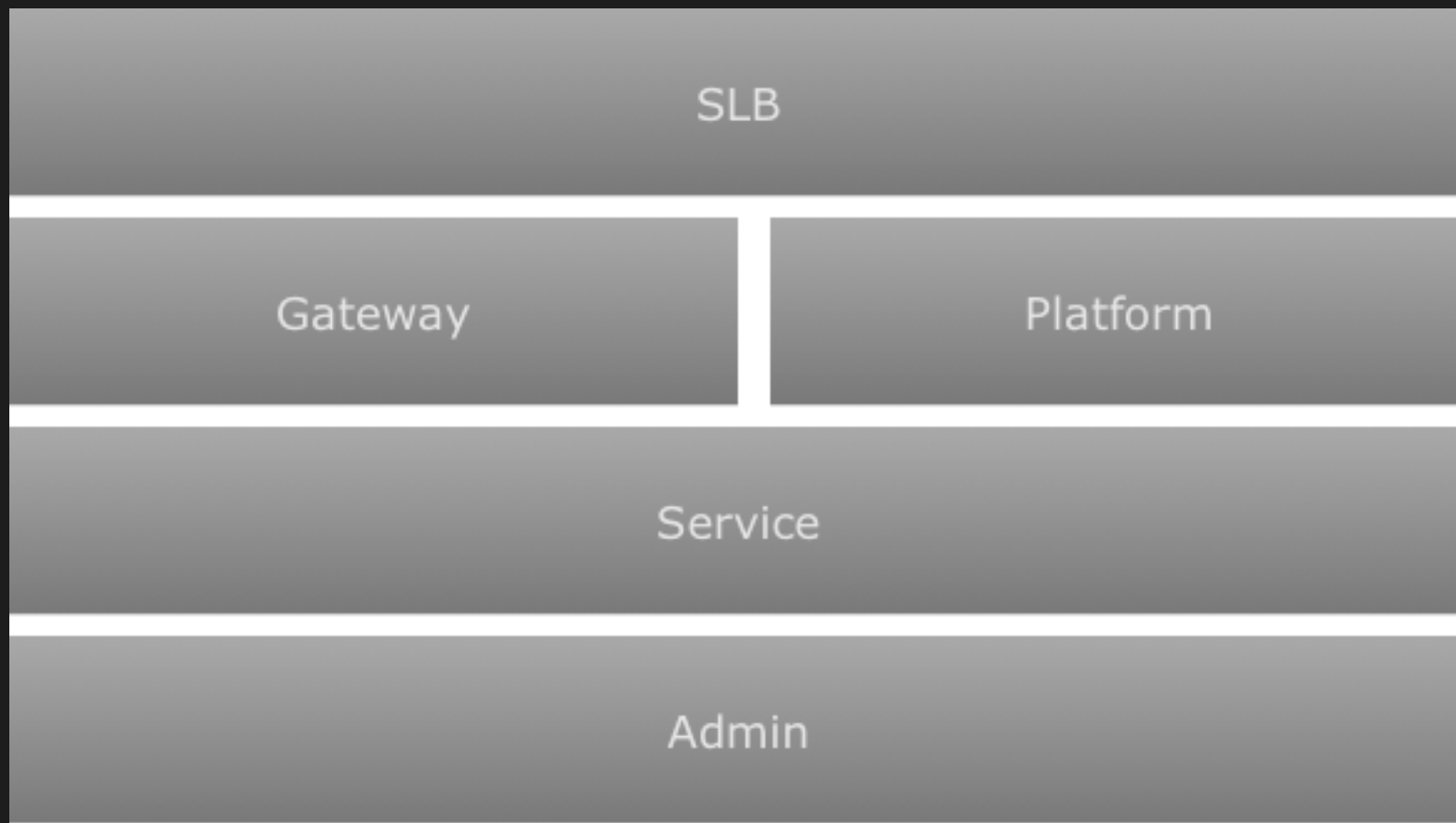
✧ 资源隔离部署

✧ 内外网服务隔离

✧ RPC框架

✧ API Gateway

微服务的演进



微服务的演进

✧梳理业务边界

✧资源隔离部署

✧内外网服务隔离

✧RPC框架

✧API Gateway

微服务的演进

- ✧序列化 (GOB)
- ✧上下文管理 (超时控制)
- ✧拦截器 (鉴权、统计、限流)
- ✧服务注册 (Zookeeper)
- ✧负载均衡 (客户端)

微服务的演进

```
type TestArgs struct {
    A, B int
}

type TestReply struct {
    C int
}

type TestTimeout struct {
    T time.Duration
}

type TestRPC int


func (t *TestRPC) Add(c context.Context, args *TestArgs, reply *TestReply) error {
    reply.C = args.A + args.B
    return nil
}

func (t *TestRPC) Timeout(c context.Context, args *TestTimeout, reply *struct{}) error {
    log.Printf("Timeout: timeout=%s, seq=%d\n", args.T, c.Seq())
    time.Sleep(args.T)
    return nil
}
```

微服务的演进

```
// First arg need not be a pointer.  
argType := mtype.In(1)  
if !argType.Implements(ctxType) {  
    if reportErr {  
        log.Println(mname, "argument type must implements:", ctxType)  
    }  
    continue  
}
```

```
var (  
    // Precompute the reflect type for error. Can't use error directly  
    // because Typeof takes an empty interface value. This is annoying.  
    typeOfError = reflect.TypeOf((*error)(nil)).Elem()  
    ctxType     = reflect.TypeOf((*context.Context)(nil)).Elem()  
    class       = trace.ClassService  
  
    _pingArg = &struct{}{}  
)
```



微服务的演进

```
// Context web context interface
type Context interface {
    ctx.Context
    Now() time.Time
    Seq() uint64
    ServiceMethod() string
    User() string
}

// rpcCtx only used in srpc.
type rpcCtx struct {
    ctx.Context
    now          time.Time
    seq          uint64
    serviceMethod string
    user         string
}

// NewContext new a rpc context.
func NewContext(c ctx.Context, u, m string, s uint64) Context {
    rc := &rpcCtx{Context: c, now: time.Now(), seq: s, serviceMethod: m, user: u}
    return rc
}
```

微服务的演进

```
// Interceptor interface.
type Interceptor interface {
    Rate(context.Context) error
    Stat(context.Context, interface{}), error)
    Auth(context.Context, net.Addr, string) error // ip, token
}
```

```
if server.Interceptor != nil {
    if req.Trace != nil {
        c1 = trace.NewContext2(c1, req.Trace)
    }
    req.ctx = context.NewContext(c1, codec.auth.User, req.ServiceMethod, req.Seq)
    if err = server.Interceptor.Auth(req.ctx, codec.addr, codec.auth.Token); err != nil {
        errmsg = err.Error()
    }
    server.sendResponse(req.ctx, codec, invalidRequest, errmsg)
}
return
```

微服务的演进

```
// ServeConn runs the server on a single connection.
// ServeConn blocks, serving the connection until the client hangs up.
// The caller typically invokes ServeConn in a go statement.
// ServeConn uses the gob wire format (see package gob) on the
// connection. To use an alternate codec, use ServeCodec.
func (server *Server) ServeConn(conn io.ReadWriteCloser) {
    buf := bufio.NewWriter(conn)
    srv := &gobServerCodec{
        rwc:    conn,
        dec:    gob.NewDecoder(conn),
        enc:    gob.NewEncoder(buf),
        encBuf: buf,
    }
    server.ServeCodec(srv)
}
```

```
type serverCodec struct {
    sending sync.Mutex
    resp    Response
    req     Request
    auth    Auth

    rwc    io.ReadWriteCloser
    dec    *gob.Decoder
    enc    *gob.Encoder
    encBuf *bufio.Writer
    addr   net.Addr
    closed bool
}
```


微服务的演进

```
func (server *Server) getRequest() *Request {
    server.reqLock.Lock()
    req := server.freeReq
    if req == nil {
        req = new(Request)
    } else {
        server.freeReq = req.next
        *req = Request{}
    }
    server.reqLock.Unlock()
    return req
}
```

```
func (server *Server) freeRequest(req *Request) {
    server.reqLock.Lock()
    req.next = server.freeReq
    server.freeReq = req
    server.reqLock.Unlock()
}
```

```
// serveCodec is like ServeConn but uses the specified codec to
// decode requests and encode responses.
func (server *Server) serveCodec(codec *serverCodec) {
    req := &codec.req
    if err := server.handshake(codec); err != nil {
        codec.close()
        return
    }
    for {
```

微服务的演进

```
// RPCServer rpc server settings.
type RPCServer struct {
    Proto string
    Addr  string
    Group string
    Weight int // weight of rpc server and also means num of client connections.
}

// balancer interface
type balancer interface {
    Boardcast(context.Context, string, interface{}, interface{}) error
    Call(context.Context, string, interface{}, interface{}) error
    SetMethodTimeout(method string, timeout time.Duration)
    SetTimeout(timeout time.Duration)
}

// wrr get available rpc client by wrr strategy.
type wrr struct {
    pool []*clients
    weight int64
    server int64
    idx int64
}
```

微服务的演进

- ✧ 梳理业务边界
- ✧ 资源隔离部署
- ✧ 内外网服务隔离
- ✧ RPC框架
- ✧ API Gateway

微服务的演进

✧统一&聚合协议

✧errgroup并行调用

✧业务隔离

✧熔断、降级、限流等高可用

Agenda

✧ 微服务的演进

✧ 高可用

✧ 中间件

✧ 持续集成和交付

✧ 运维体系

高可用

✧ 隔离

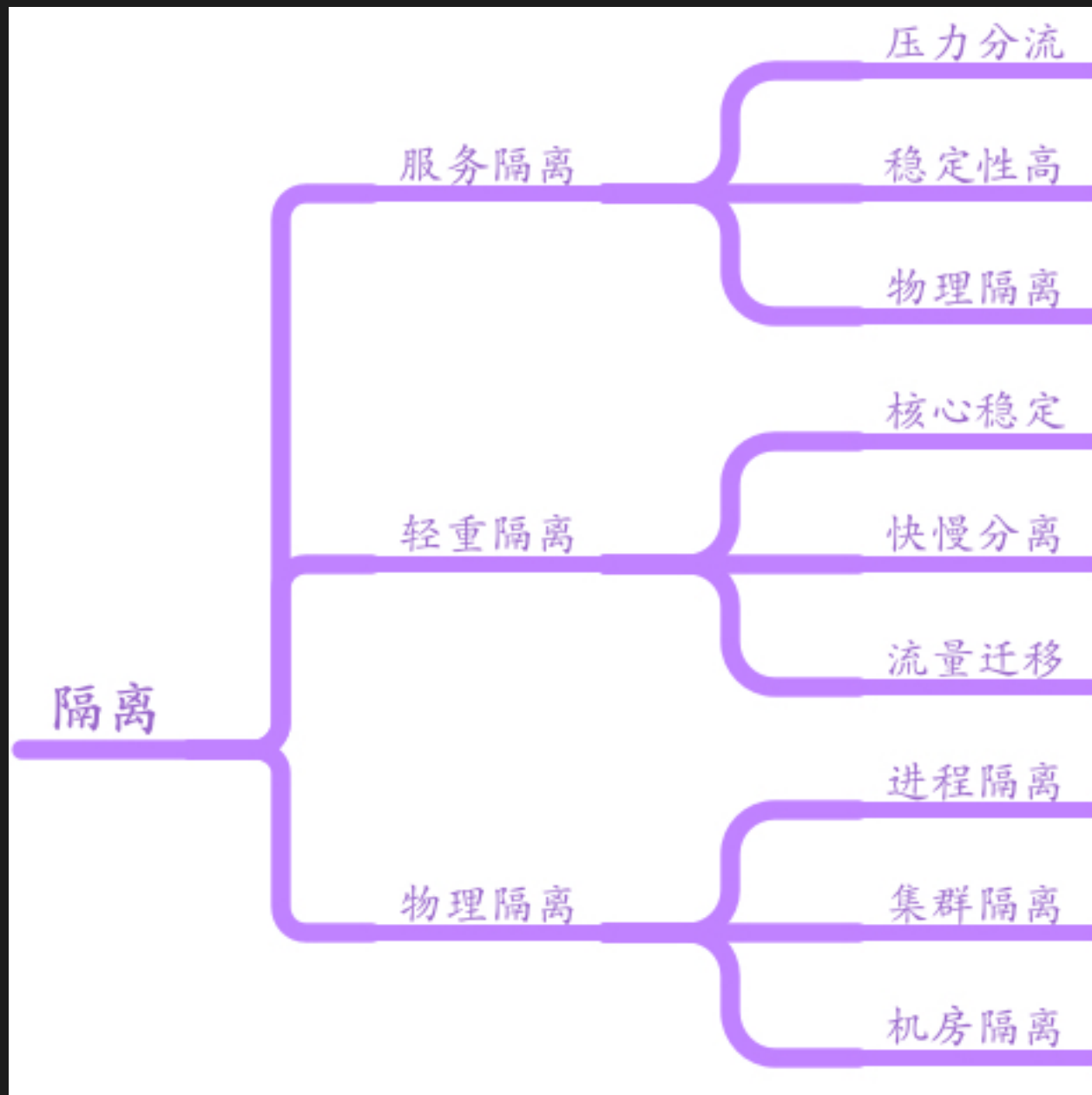
✧ 超时

✧ 限流

✧ 降级

✧ 容错

高可用



高可用

◇ 隔离

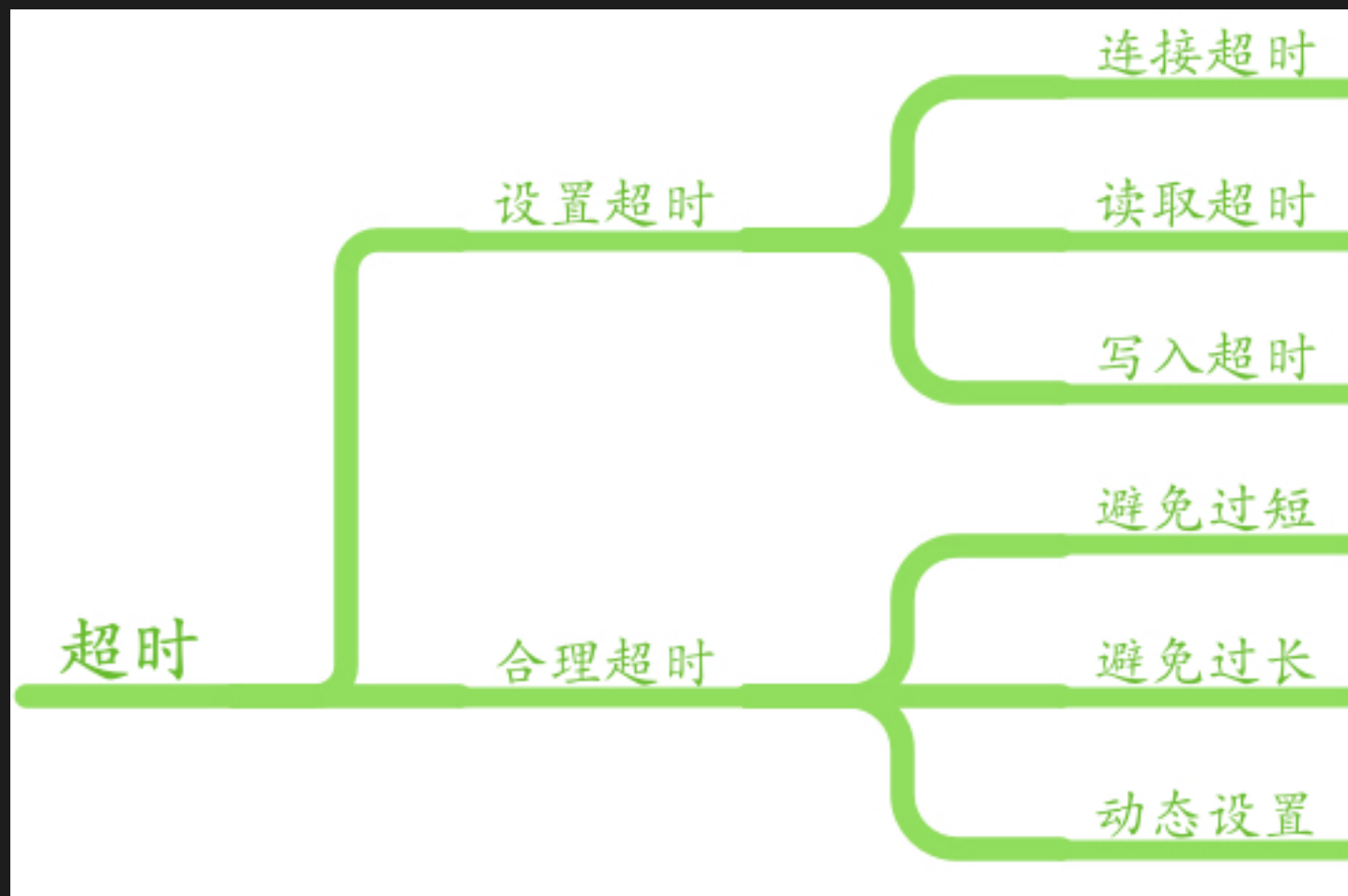
◇ 超时

◇ 限流

◇ 降级

◇ 容错

高可用



高可用

✧ 隔离

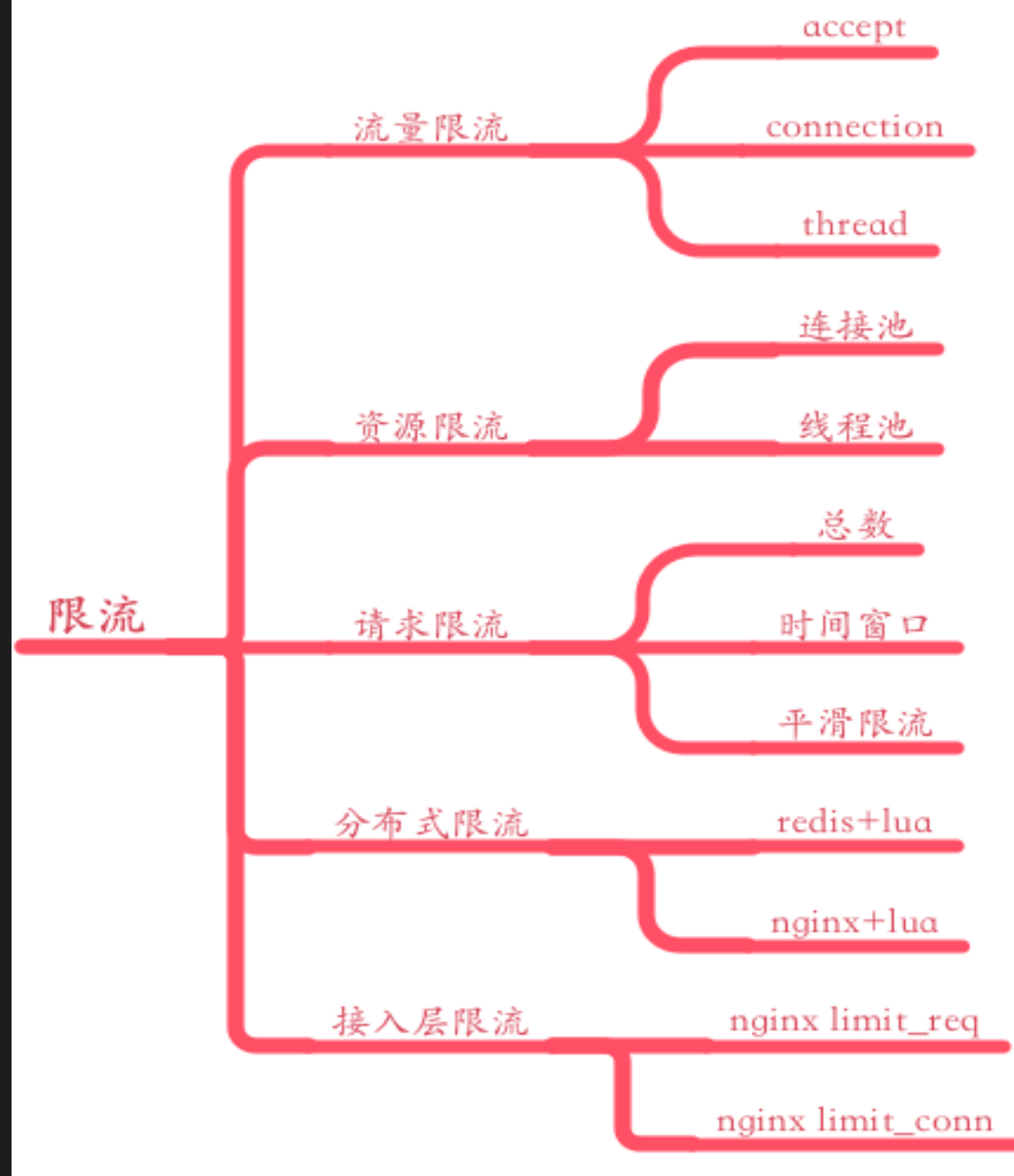
✧ 超时

✧ 限流

✧ 降级

✧ 容错

高可用



高可用

✧ 隔离

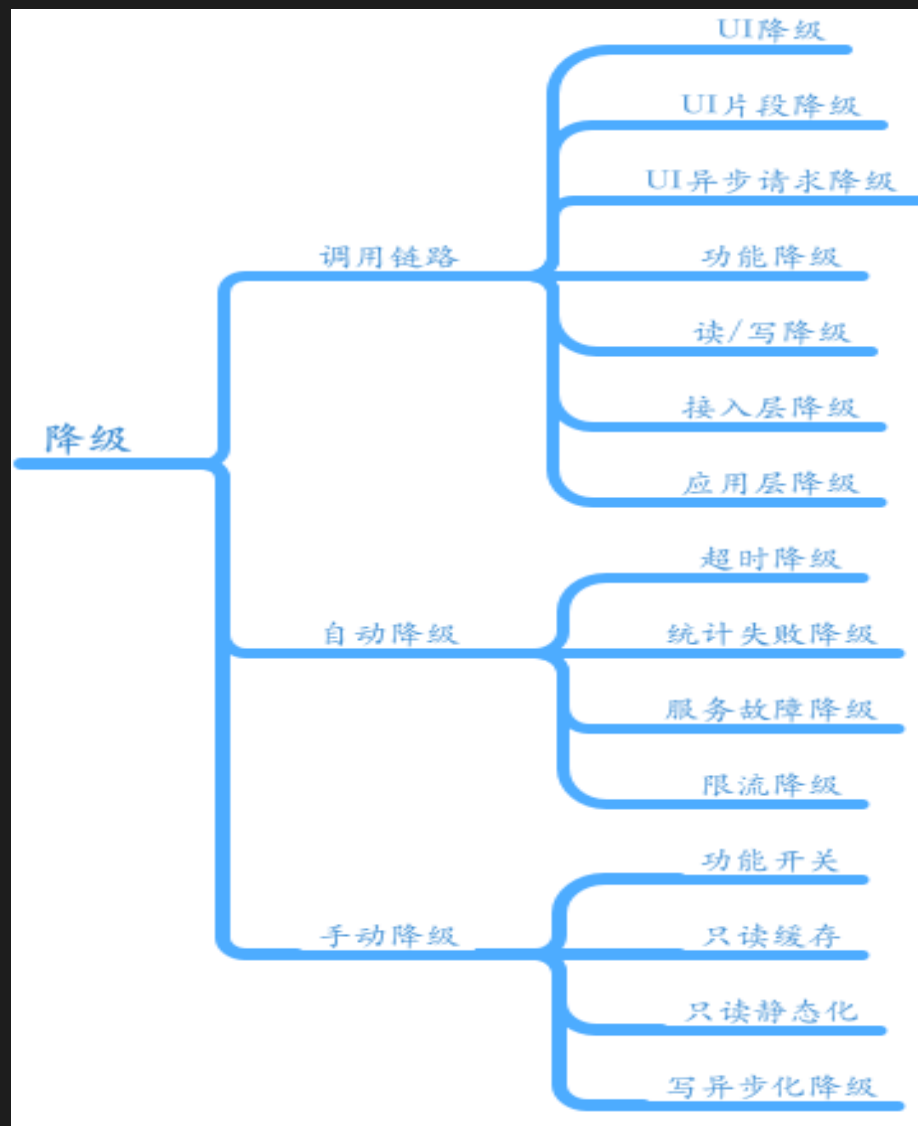
✧ 超时

✧ 限流

✧ 降级

✧ 容错

高可用



高可用

✧ 隔离

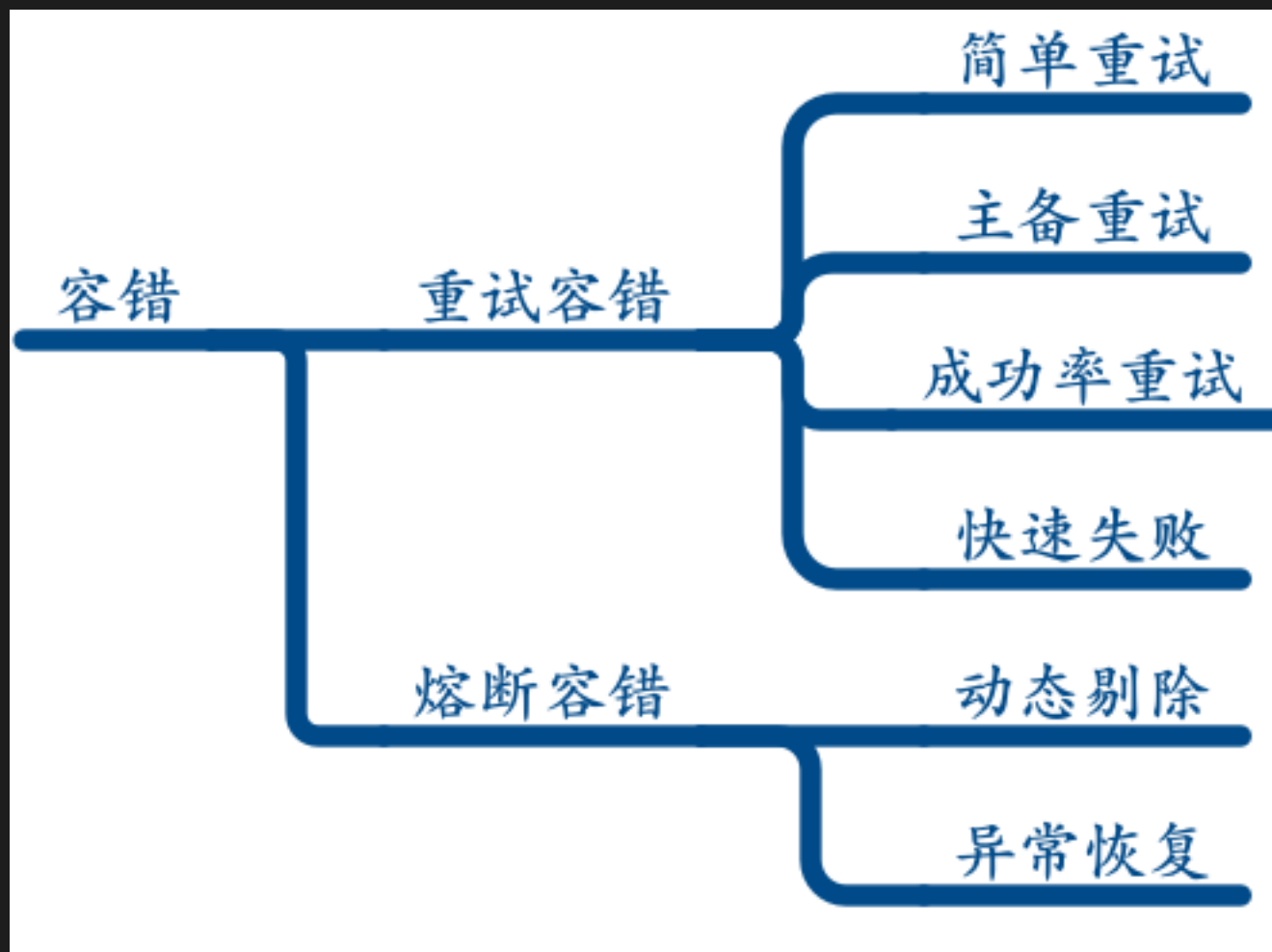
✧ 超时

✧ 限流

✧ 降级

✧ 容错

高可用



Agenda

✧ 微服务的演进

✧ 高可用

✧ 中间件

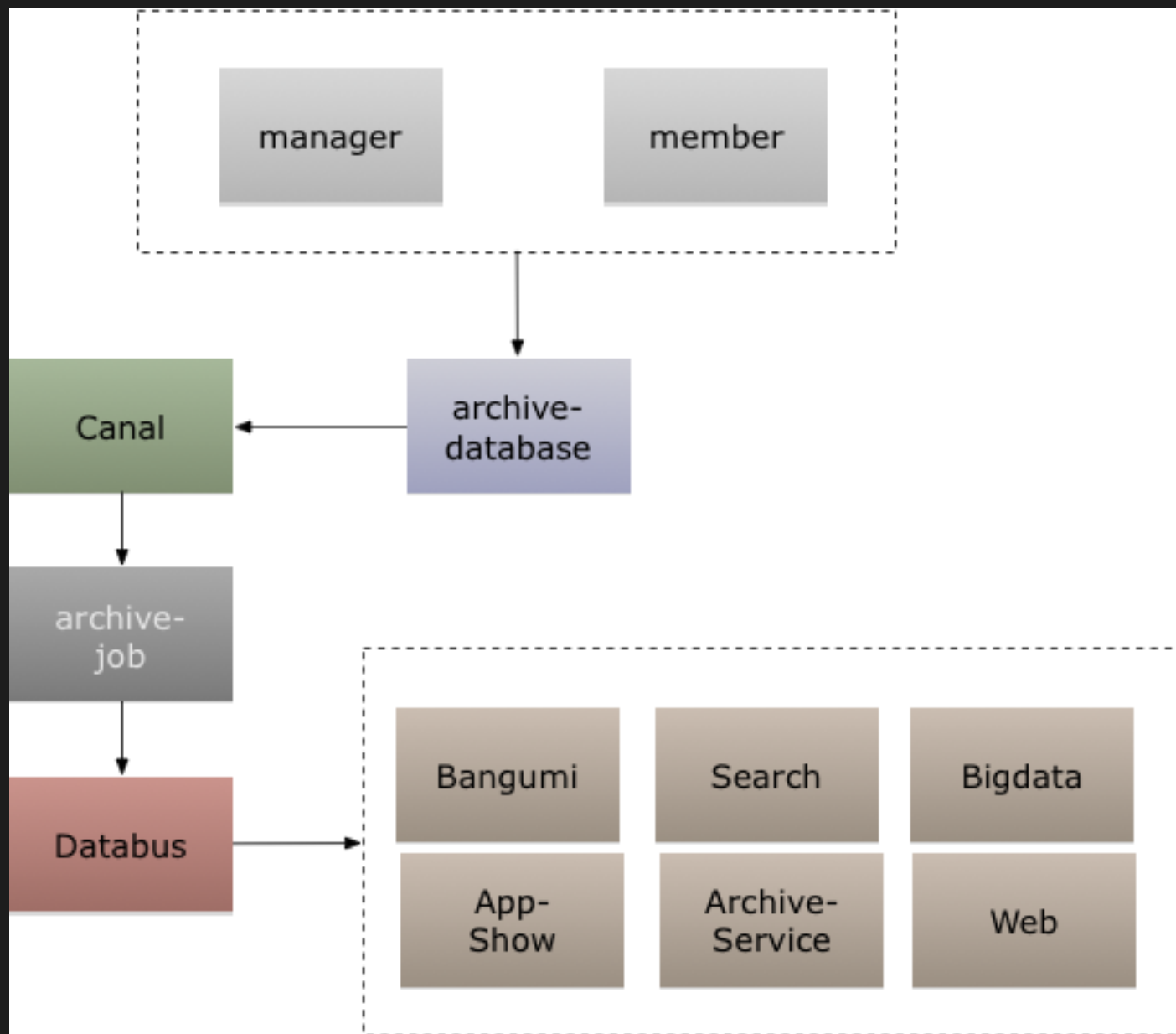
✧ 持续集成和交付

✧ 运维体系

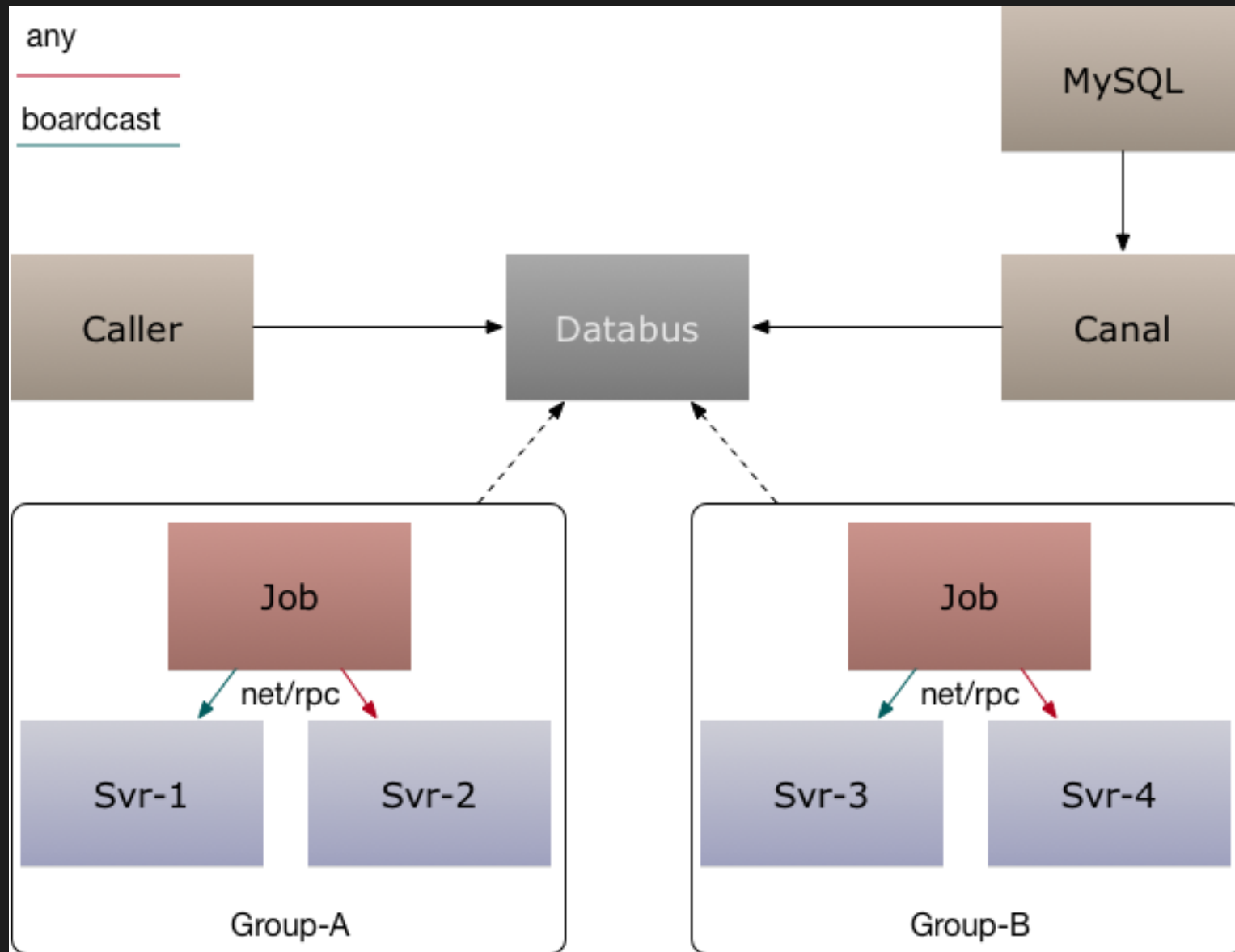
中间件

- ✧ databus (基于Kafka)
- ✧ canal (MySQL Replication)
- ✧ bilitw (基于Twemproxy)
- ✧ bfs (facebook haystack, opencv)
- ✧ config-service
- ✧ dapper (google dapper)

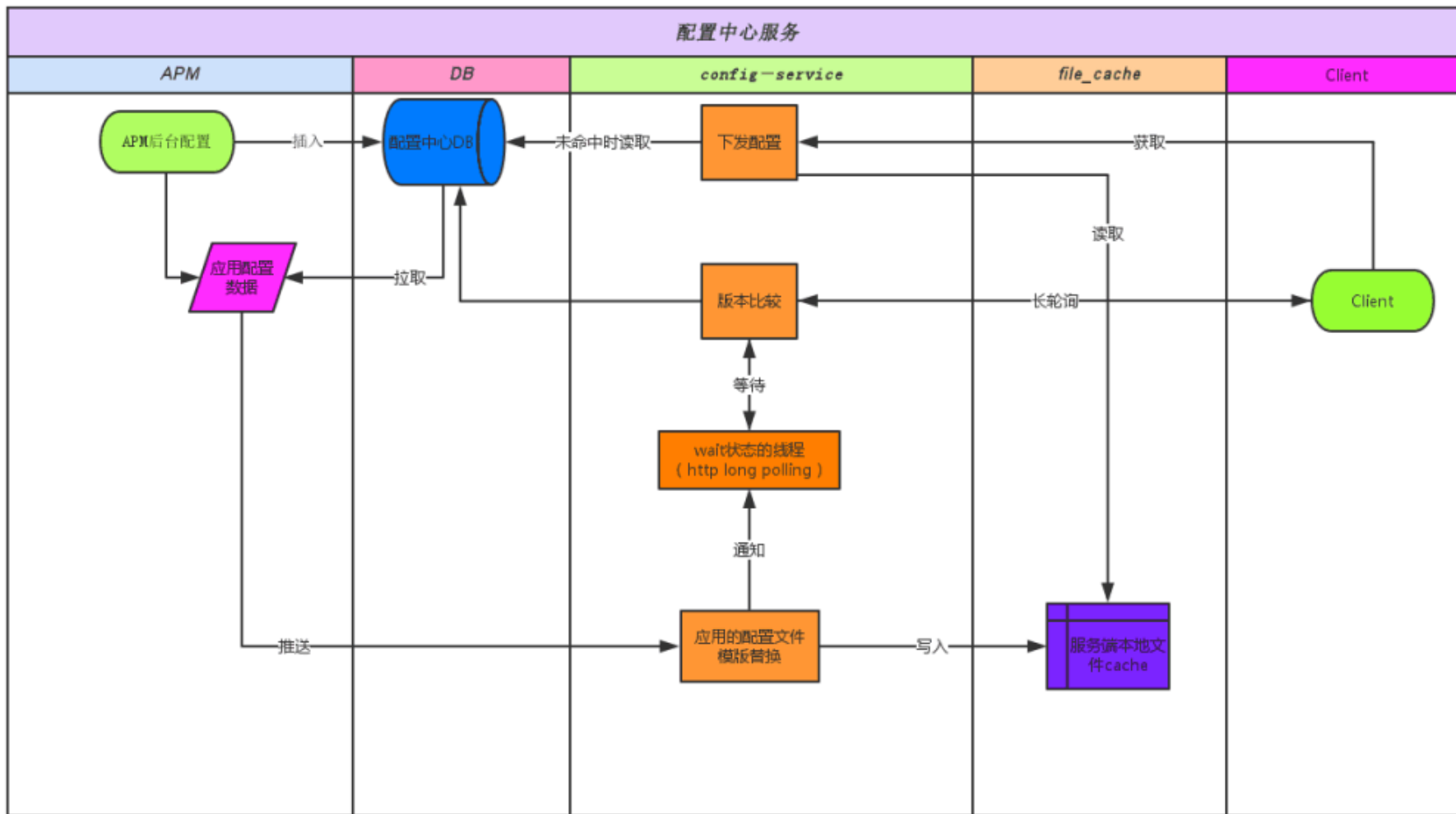
中间件



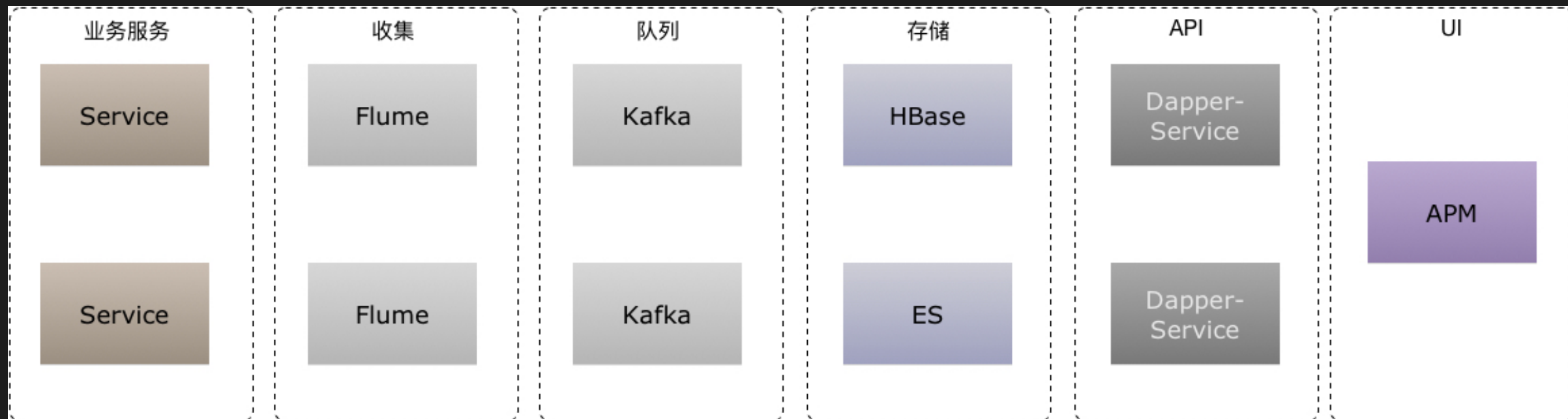
中间件



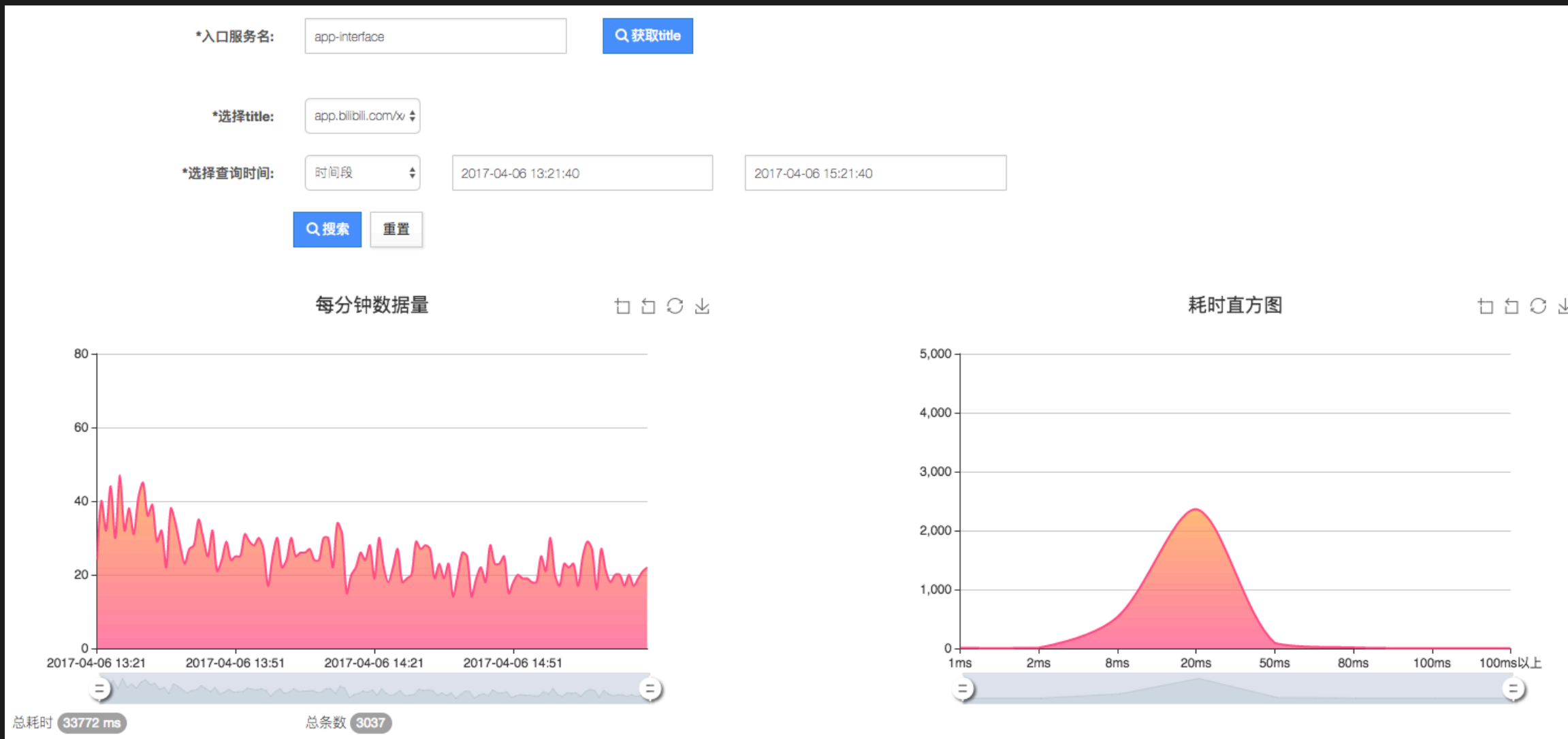
中间件



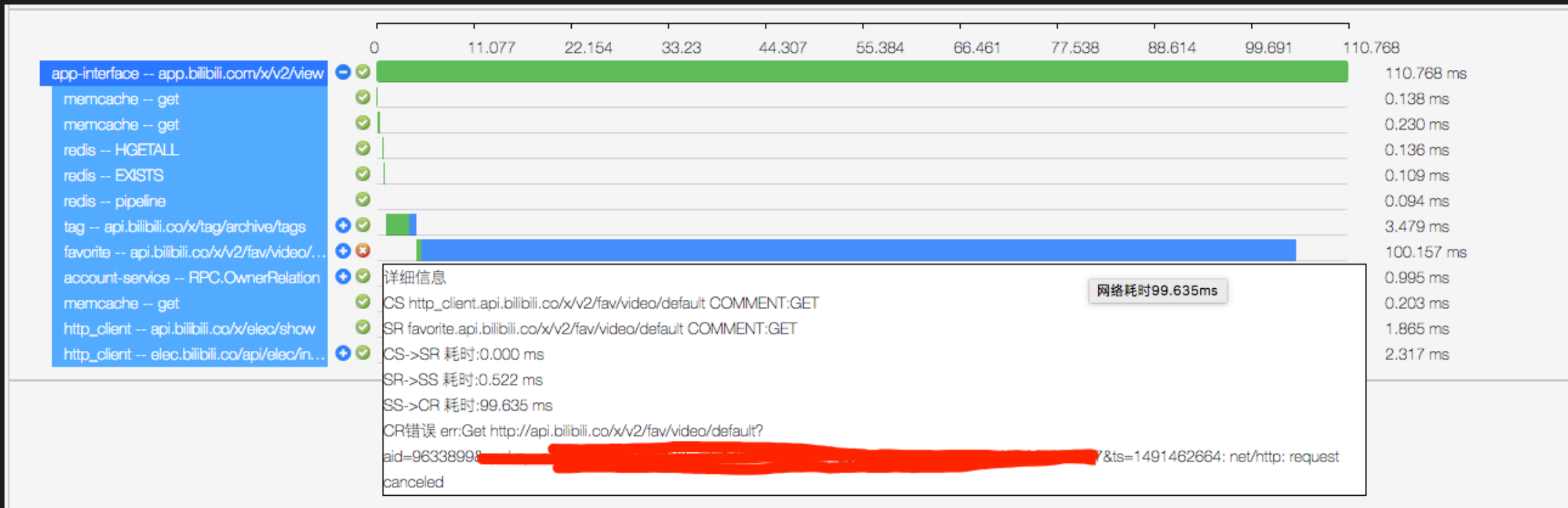
中间件



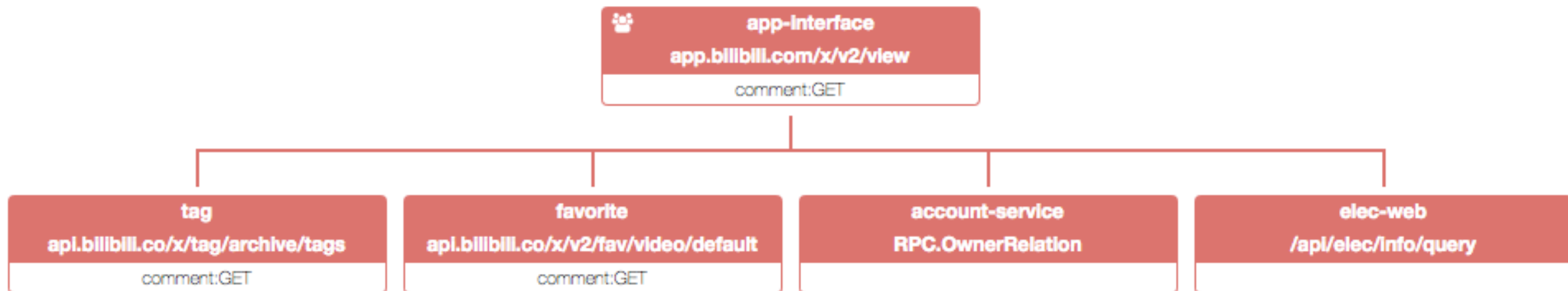
中间件



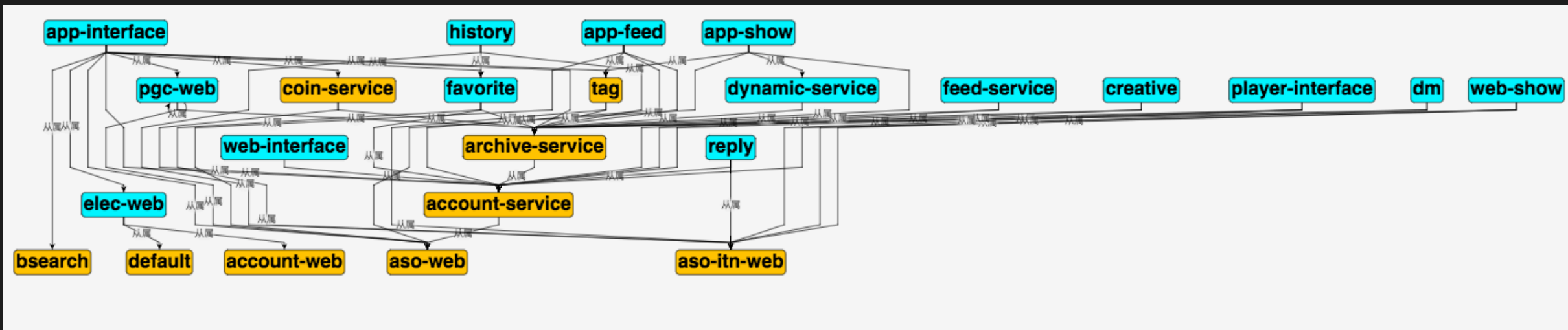
中间件



中间件



中间件



中间件

```
// Request is a header written before every RPC call. It is used internally
// but documented here as an aid to debugging, such as when analyzing
// network traffic.
type Request struct {
    ServiceMethod string // format: "Service.Method"
    Seq            uint64 // sequence number chosen by client
    Trace         *trace.Trace2 // trace info

    ctx context.Context
}

// WithHTTP set trace id into http request.
func (t *Trace2) WithHTTP(req *http.Request) {
    req.Header.Set(_httpHeaderID, strconv.FormatUint(t.ID, 10))
    req.Header.Set(_httpHeaderSpanID, strconv.FormatUint(t.SpanID, 10))
    req.Header.Set(_httpHeaderParentID, strconv.FormatUint(t.ParentID, 10))
    req.Header.Set(_httpHeaderSampled, strconv.FormatBool(t.Sampled))
    req.Header.Set(_httpHeaderLevel, strconv.FormatInt(int64(t.Level), 10))
    req.Header.Set(_httpHeaderUser, Owner())
}
```

Agenda

✧ 微服务的演进

✧ 高可用

✧ 中间件

✧ 持续集成和交付

✧ 运维体系

持续集成和交付

- ✧ 版本管理（语义化）
- ✧ 分支管理（gitlab+mr review）
- ✧ 环境管理（集成环境）
- ✧ 测试（单元测试，服务测试）
- ✧ 发布（冒烟、灰度、蓝绿）

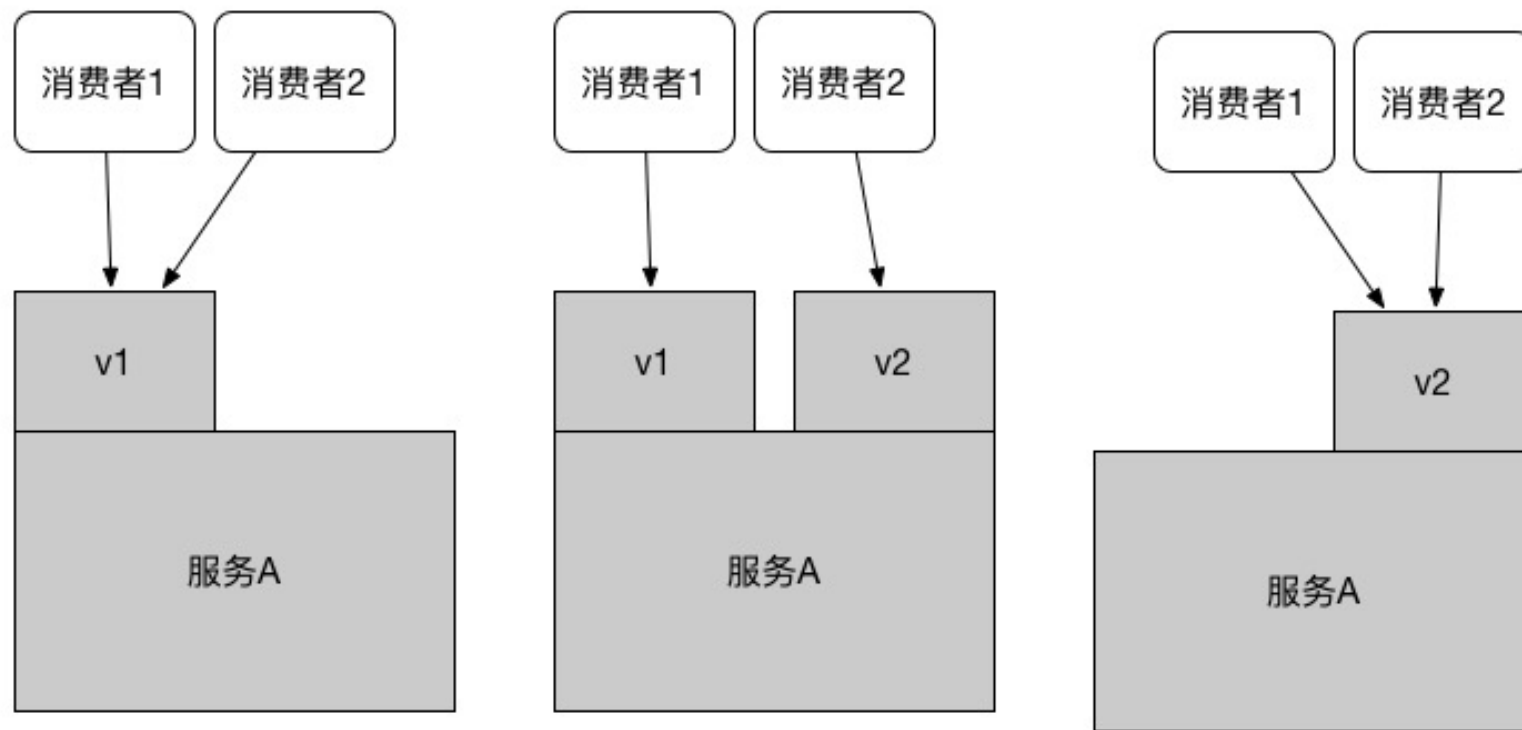
持续集成和交付

使用语义化的版本管理
MAJOR.MINOR.PATCH

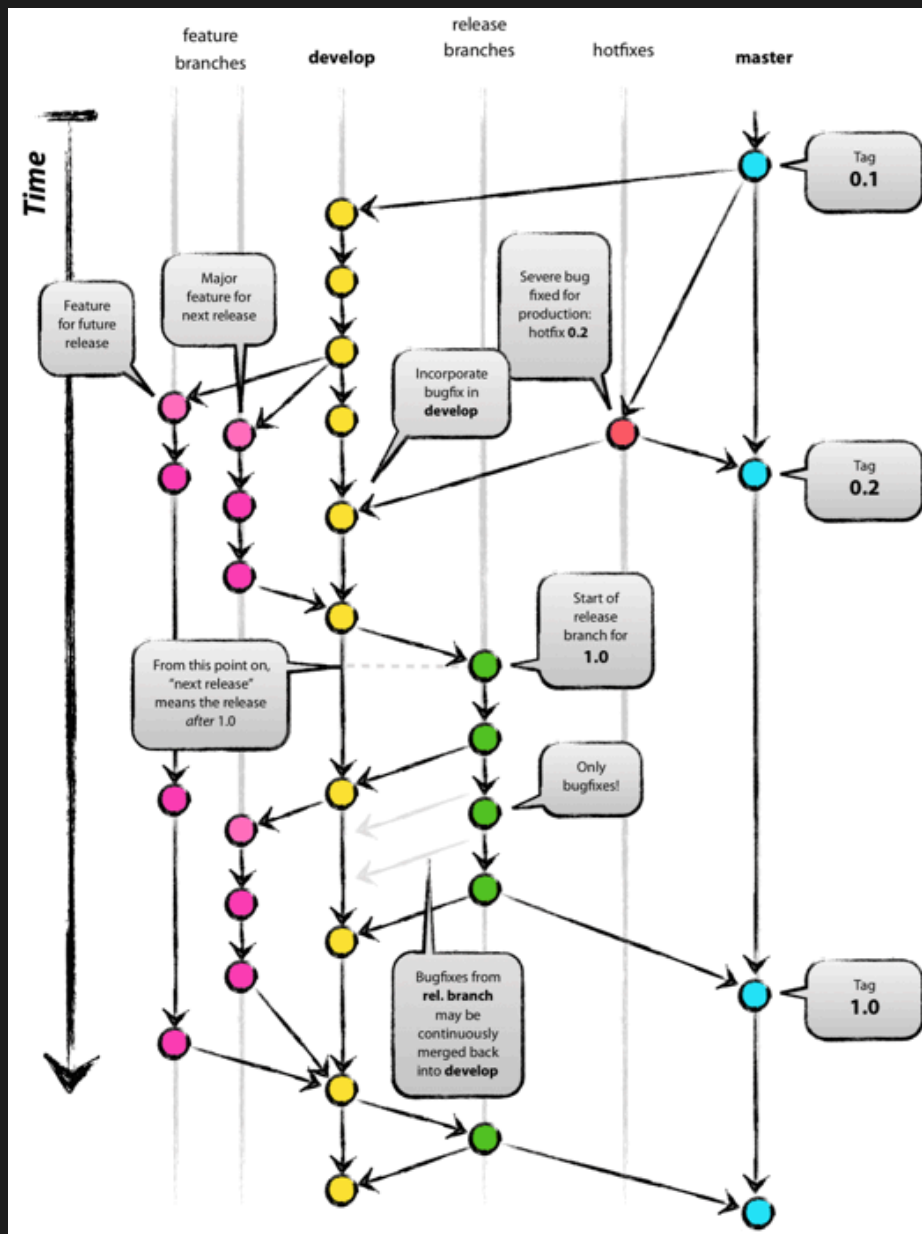
- ✧ MAJOR：改变意味着其中包含不向后兼容的修改；
- ✧ MINOR：改变意味着有新功能的增加，但应该是向后兼容的；
- ✧ PATCH：改变代表对已有功能的bug修复；

因此是使用对方服务时候，需要明确有微服务或者是API的版本管理，基于此我们知道是否是兼容的。

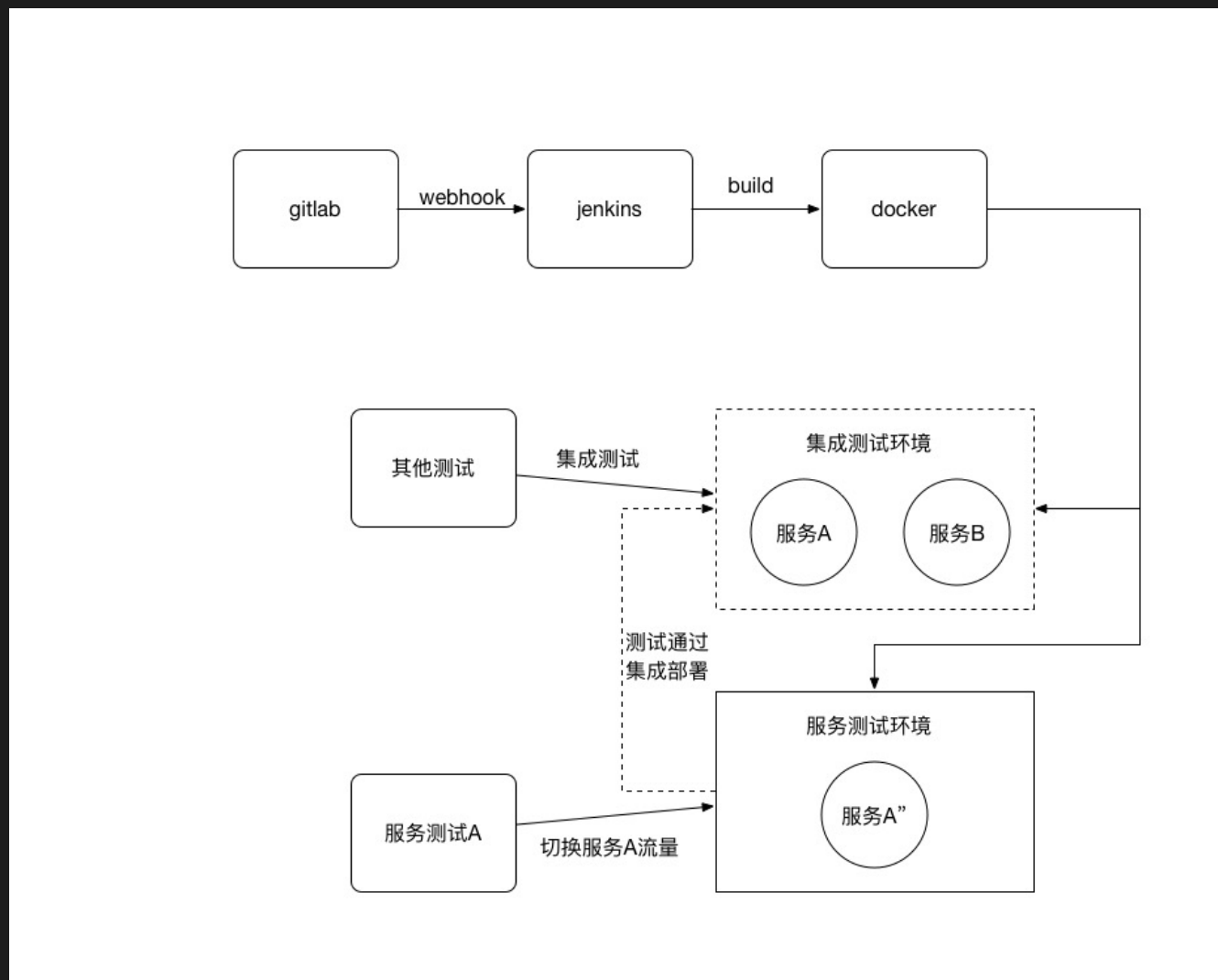
持续集成和交付



持续集成和交付



持续集成和交付



持续集成和交付



Agenda

- ◇ 微服务的演进
- ◇ 高可用
- ◇ 中间件
- ◇ 持续集成和交付
- ◇ 运维体系

运维体系

APP信息

应用ID	部署环境	描述	Pool归属	Image	版本
account-service	prod	主站账号缓存服务	platform(主站app)	docker-reg.bilibili.co/account-service(主站账号缓存服务)	v5.6.0

容器情况

正常更新 灰度更新 重启

环境	IP	Port	宿主机	Image	配置中心版本	Status	Health	启动时间	
生产	172.18.62.172	6071		docker-reg.bilibili.co/account-service:v5.6.0	v5-docker	启动	health	2017-03-21 15:30:03	控制台 日志 实时监控
生产	172.18.62.88	6071		docker-reg.bilibili.co/account-service:v5.6.0	v5-docker	启动	health	2017-03-17 17:53:43	控制台 日志 实时监控
生产	172.18.62.193	6071		docker-reg.bilibili.co/account-service:v5.6.0	v5-docker	启动	health	2017-03-21 15:29:51	控制台 日志 实时监控
生产	172.18.62.149	6071		docker-reg.bilibili.co/account-service:v5.6.0	v5-docker	启动	health	2017-03-17 17:54:02	控制台 日志 实时监控
生产	172.18.62.132	6071		docker-reg.bilibili.co/account-service:v5.6.0	v5-docker	启动	health	2017-03-21 15:29:54	控制台 日志 实时监控

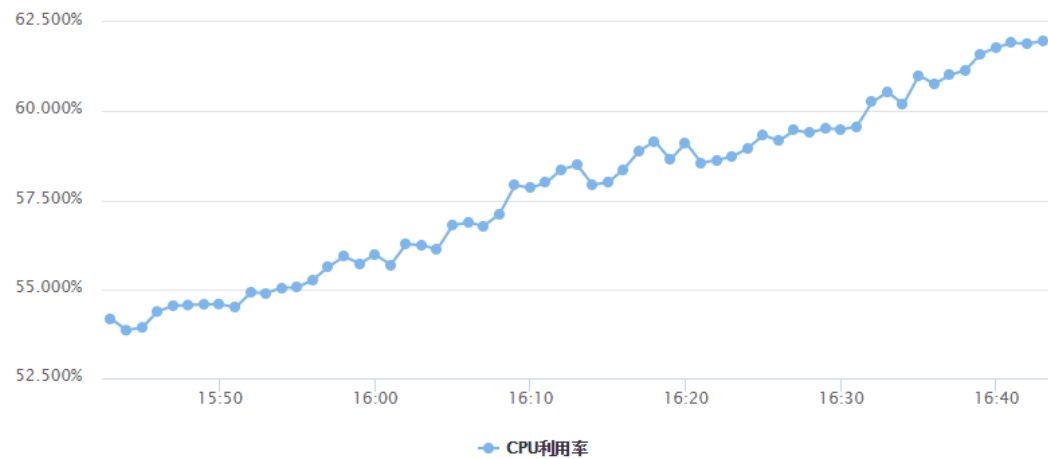
显示第 1 到第 5 条记录, 总共 8 条记录 每页显示 条记录 « < 1 2 > »

时间	用户	动作	状态	更新前版本	更新后版本
2017-03-21 15:29:48	wuanchuang	扩容	成功	-	-
2017-03-17 17:53:26	wuanchuang			-	-

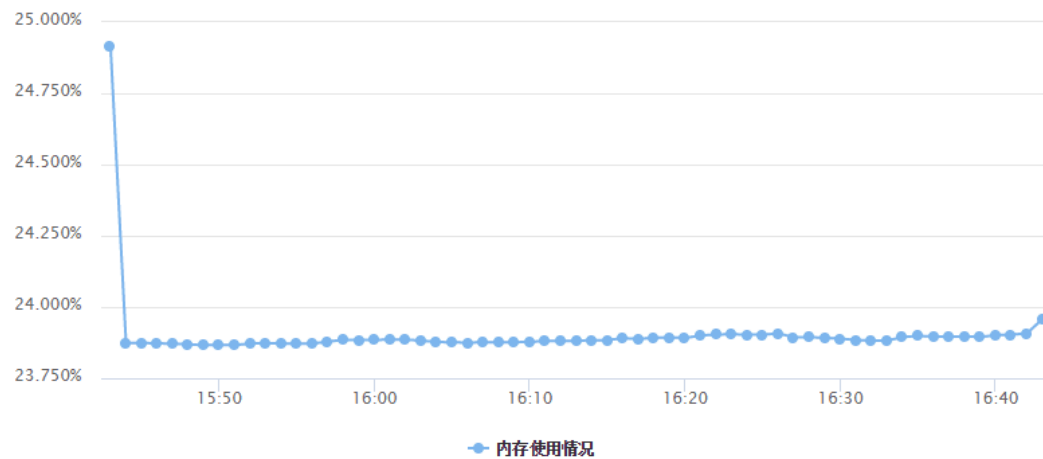
100%

运维体系

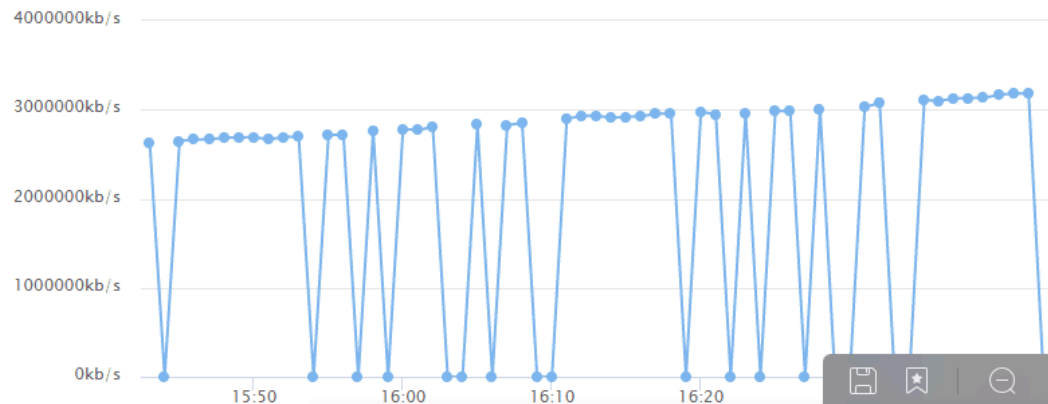
CPU利用率



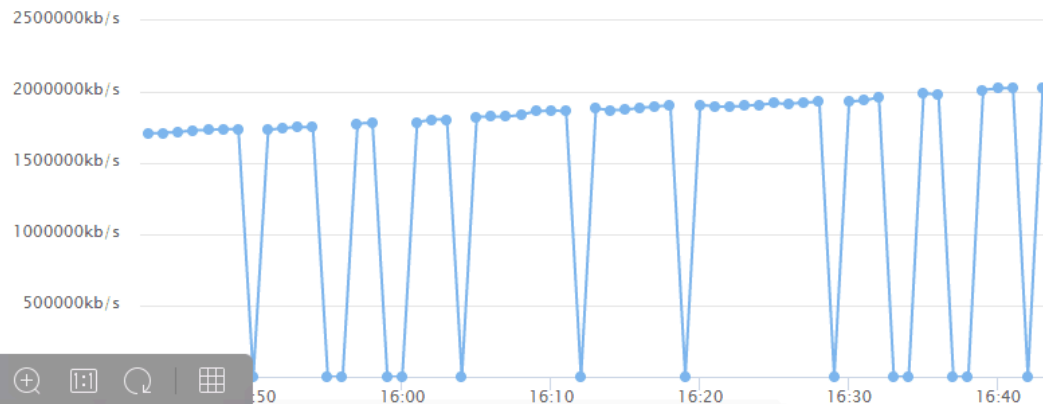
内存使用情况



网络流量 入带宽



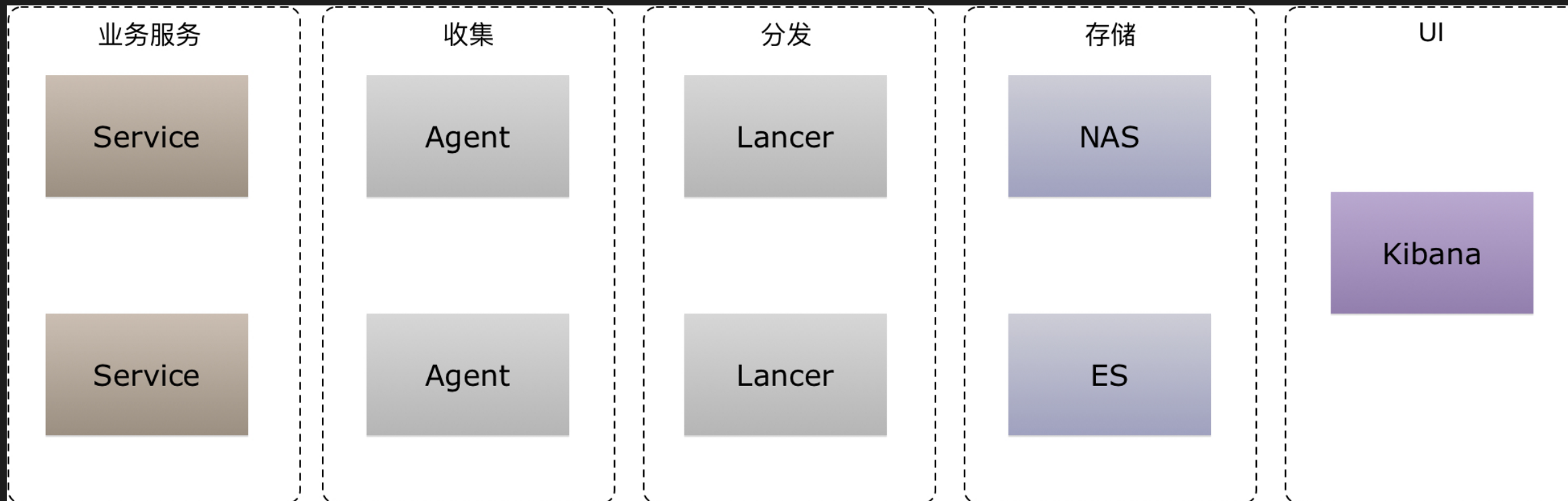
网络流量 出带宽



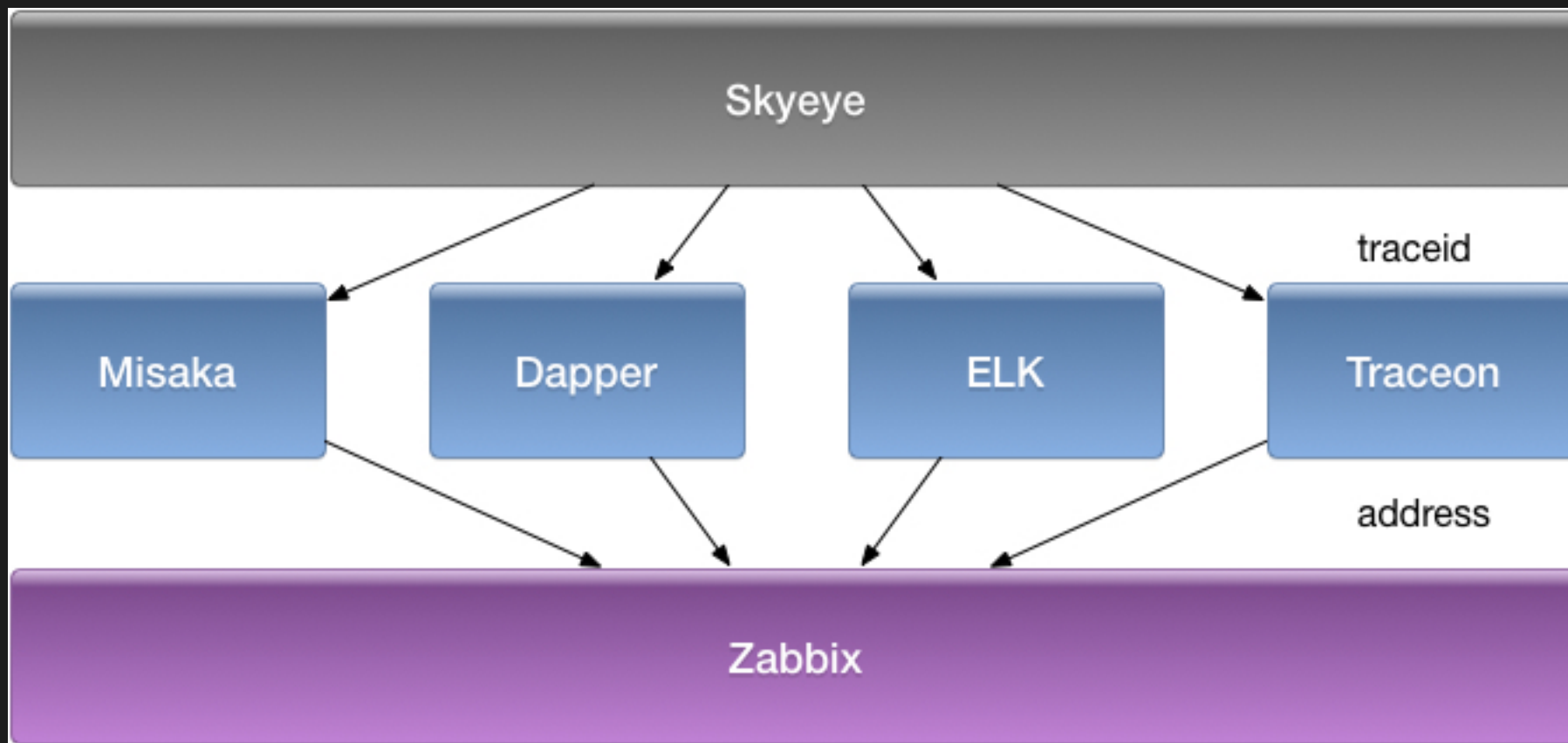
运维体系

```
欢迎使用BiliPaas的Console功能  
你已经在容器里面,小心操作啊!  
root@08b2077dd10d:/#
```

运维体系

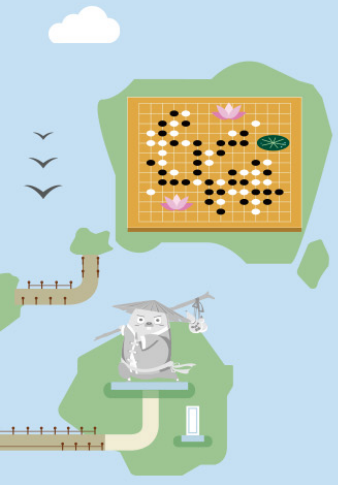


运维体系



运维体系





THANKS

——毛剑

