CloudWeGo

智谱清言

单体到微服务架构转型之路

[肖文彬-智谱AI]

2024/03/22

CONTENT
目录

01

技术变迁

初期技术背景及痛点

智谱清言是谁

基于自主研发的GLM模型的全能AI助手

国内首批上线的大模型产品之一，赢得市场广泛认可

不受地点限制，您可以随时与强大的GLM-4 大模型进行对话

无需编程知识

不需要有任何编程背景，通过自然对话就可以使用这个APP

全方位满足用户需求

多平台支持

不论是手机APP、电脑还是微信小程序，都能轻松使用

零代码创建智能体

轻松创建自己的智能体，无论是用于提升生产力、娱乐还是个性化需求

自动理解，规划复杂指令

扫码立即体验

项目初期

敏捷开发，快速迭代

国内首批上线，时间紧迫，项目与架构相对粗放

项目架构

使用Python(Flask)构建的单体应用

开发资源短缺

初期只有四名后端同学，大家除了开发业务，环境、发布、测试一把抓

除了基础的硬件资源，各种都要从零做起

项目耦合

所有功能集中在同个项目，难以复用

单体应用与微服务对比

	单体应用	微服务
可伸缩性	当需要扩展特定功能或模块时，必须扩展整个应用，这可能导致资源浪费	可以独立扩展需要更多资源的特定服务，而无需扩展整个应用，从而更有效地利用资源
灵活性	更改任何部分都可能需要重新部署整个应用，且不同团队可能互相阻塞	每个服务可以独立部署和更新，允许更快地迭代和更灵活的开发流程
容错性	一个组件的失败可能导致整个应用崩溃	一个服务的失败通常只会影响该服务，而不会影响到其他服务，从而提高系统的整体稳定性
技术多样性	通常整个应用使用相同的技术栈	每个服务可以采用最适合其业务需求的技术栈，促进技术创新
团队自治	大型团队可能需要协调工作，导致效率低下	小型、跨功能的团队可以独立负责自己的服务，提高开发效率
可维护性	随着应用的增长，代码库可能变得庞大且复杂，难以理解和维护	服务通常是小的、集中的，更容易理解和维护
部署频率	由于规模和复杂性，部署可能是一个耗时且风险较高的过程	可以频繁且独立地部署服务，减少部署风险

微服务挑战

服务拆分

确定服务的边界和粒度是一个复杂的过程，拆分不当可能导致过度耦合或过度分散的服务

分布式系统复杂性

微服务架构本质上是分布式的，这增加了系统的复杂性

包括网络延迟、容错、数据一致性和服务发现等问题

服务间通信

微服务之间需要通过网络进行通信，这可能导致性能问题、通信故障和数据安全问题

服务发现和配置管理

服务发现机制需要能够动态地注册和发现服务实例

配置管理则需要确保服务实例能够获取正确的配置信息

微服务升级

确定技术栈

使用Golang作为新语言，天生支持高并发与低延迟，适合作为后端语言

开发公共通用SDK，新服务需要引入

服务拆分

旧服务优先拆分用户登录鉴权，新服务保持原响应结构，通过网关将路由指向新服务。数个版本后彻底迁移

新业务直接在新服务中开发

独立存储资源

各微服务采用独立的数据库、缓存资源，即使使用相同的存储，也要按不同的实例连接来处理

中心化配置管理

不将所有配置存放在环境变量中，放在配置中心管理

编排测试

自动化的编排部署，编写单元测试

02

微服务升级实践

技术选型及框架搭建

框架选型

Gin vs Hertz

Gin是轻量级高性能HTTP框架，几乎是Gopher的首选

Hertz参考了开源框架优势，具有高易用性、高性能、高扩展等特点

gRPC vs Kitex

gRPC使用Protobuf作为描述语言，基于HTTP/2协议，支持Unary/流式调用

Kitex支持更多的描述语言、更多的通讯协议，针对Golang语言特性进行优化

Thrift vs Protobuf

字节内部优先用thrift，在Kitex中支持更好

Protobuf在其他公司更加广泛，在Python中调用更加方便，可以直接用Postman调试

Why CloudWeGo

优秀的性能与插件

基于字节netpoll网络库，benchmark的表现更佳，在高并发环境下有更极致的性能

提供服务治理功能，支持注册发现、负载均衡、限流熔断降级等特性

高度模块化，易于扩展维护，社区提供丰富的中间件与插件

注重兼容性

对于习惯Gin/gRPC的开发者，可以无痛切换至Hertz/Kitex，可以几乎保持原有写法

支持的协议更多，并且自动嗅探，无需手动指定

提供脚手架，一键生成模版代码

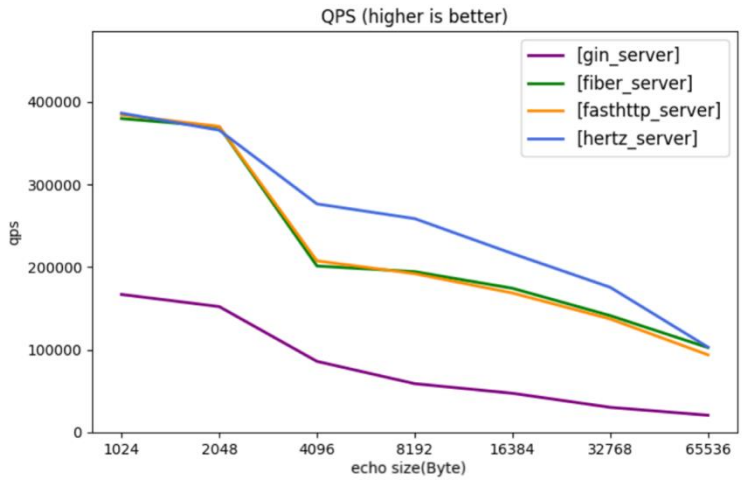
社区团队支持

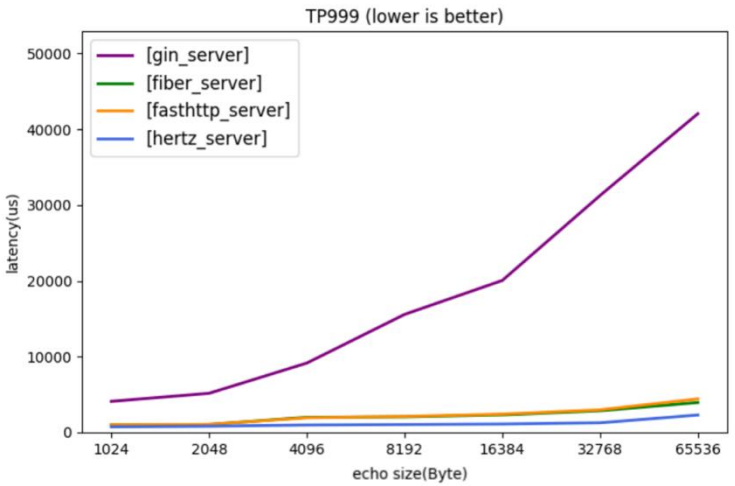
虽然Gin/gRPC都属于资历更久的框架，也有更成熟的开源社区，但交流问题仍然不便

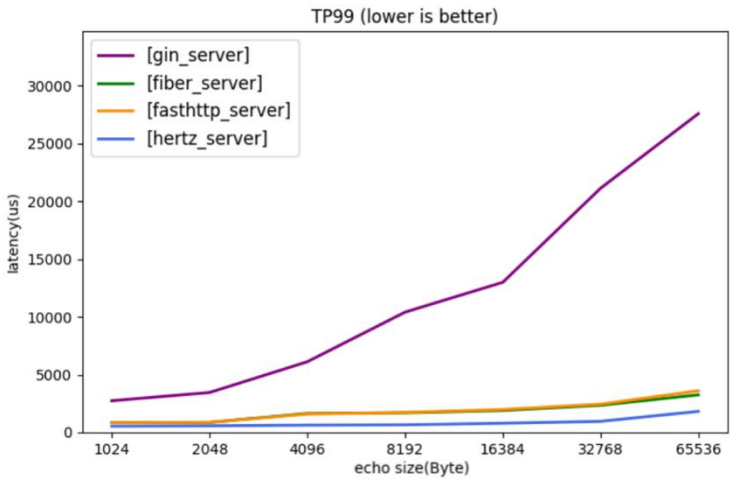
CloudWeGo为智谱专门拉了专属飞书群，响应更及时，迭代速度更快

在项目初期搭建框架时提供大力支持，互动频繁，答疑解惑

Why CloudWeGo







其他组件选型

配置中心

Nacos vs Apollo

服务中心

Nacos vs Etcd vs Consul vs Zookeeper

日志收集

SLS（三方） vs ELK

链路追踪

Arms（三方） vs Jaeger

子服务定义

Rest、AdminRest

HTTP协议，采用Hertz框架

Rest用于ToC客户端业务，AdminRest用于管理后台业务

Rpc

KitexProtobuf协议（兼容GRPC），采用Kitex框架

用于内部服务间通信

Task、Worker

常驻后台进程

Task用于单实例运行，如定时任务

Worker用于多实例运行，支持扩容，如Kafka消费者

统一的SDK

Common

通用基础组件库

提供开箱即用的常用组件，包括网络库、数据库、缓存、消息队列、日志、配置中心、工具函数等

Support

在Common库之上高度集成清言业务的封装库

Rest服务常用中间件，如登录鉴权、限流、黑名单、签名，公参注入上下文，响应结果转写

Rpc代理客户端，更友好的方式调用

IDL

protobuf、thrift等数据语言描述文件

生成的桩代码

配置中心

接口定义

```
IConfig interface {  12 个实现  xiac
    Name() string  12 个实现
    FileType() string  12 个实现
    Init()  12 个实现
}
```

```
1 个实现
rror  1 个实现
confData string)) error
1 个实现
```

支持热更新

服务启动时，从环境变量读取Nacos配置

与配置中心建立长连接进行监听

如果配置发生更改，调用ListenConfig方法reload配置实例

```
{
    "port": {
        "rest":
        "rest_prometheus":
        "rpc":
        "rpc_prometheus":
        "adminrest":
        "adminrest_prometheus":
        "health":
    },
    "discovery": true,
    "timeout": {
        "read": 60000,
        "write": 60000,
        "idle": 300000,
        "warning": 3000,
        "exit": 3000
    },
    "limit": {
        "qps": -1,
        "conn": -1
    },
    "max_body_size": 10
}
```

数据集优先级

Nacos通过namespace-group-dataid来确定唯一数据集

我们采用namespace区分服务环境，group区分服务，dataid区分不同数据集

Group优先级：服务名相同 > 仓库名相同 > default_group

入参处理

结构体定义

```
type Param struct {  无用法  新 *
    ParamStr string `json:"param_str" validate:"required" default:"aaa"`
    ParamInt int `json:"param_int" validate:"min=2" default:"10"`
}
```

参数校验

基于github.com/go-playground/validator库，针对validate的tag进行校验

原库只可以对结构体校验，无法对结构体/指针的切片/map进行校验

我们自行实现方法，基于反射递归处理，并根据default设置默认值

Protobuf注入tag

在pb的描述文件中，我们可以添加// @gotags: validate:" required"样的注释

在生成脚本中引入github.com/favadi/protoc-go-inject-tag插件，将tag注入到桩代码中

Rest结构定义

Router定义

```
type (
    // RouterGroup 分组路由，一般用于区分类名前缀
    Implement interface
    RouterGroup struct {  6 个用法  xiaowenbin
        Path        string
        Middlewares app.HandlersChain
        Groups      []*RouterGroup
        Routers     []*Router
    }

    // Router 单一路由，具体业务方法
    Implement interface
    Router struct {  3 个用法  xiaowenbin
        Method      string
        Path        string
        Middlewares app.HandlersChain
        Handler     app.HandlerFunc
    }
)
```

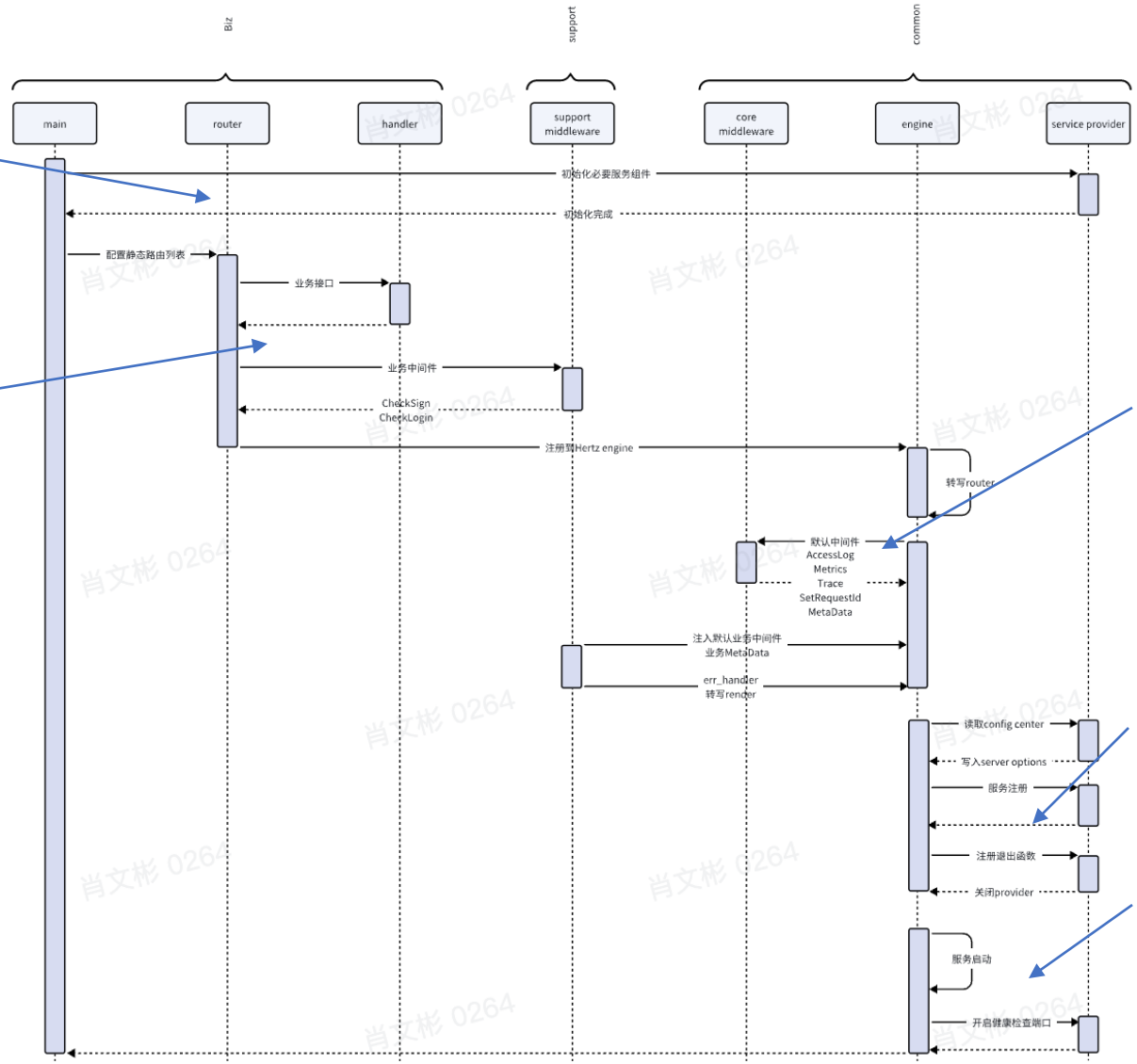
Output定义

```
IRestOutput interface {  17 个用法
    SetStatusCode(int)  1 个实现
    SetBizCode(int)  1 个实现
    SetMessage(string)  1 个实现
    SetData(any)  1 个实现
    SetError(error)  1 个实现
    SetRequestId(string)  1 个实现
    GetStatusCode() int  1 个实现
    GetBizCode() int  1 个实现
    GetMessage() string  1 个实现
    GetData() any  1 个实现
    GetError() error  1 个实现
    GetRequestId() string  1 个实现
    Clone() IRestOutput  1 个实现
}
```

Hertz启动

1. 初始化服务组件

2. 选取业务接口、中间件
配置路由并注册



3. 加入默认前置中间件
设置err_handler、render
将路由转写为engine可识别

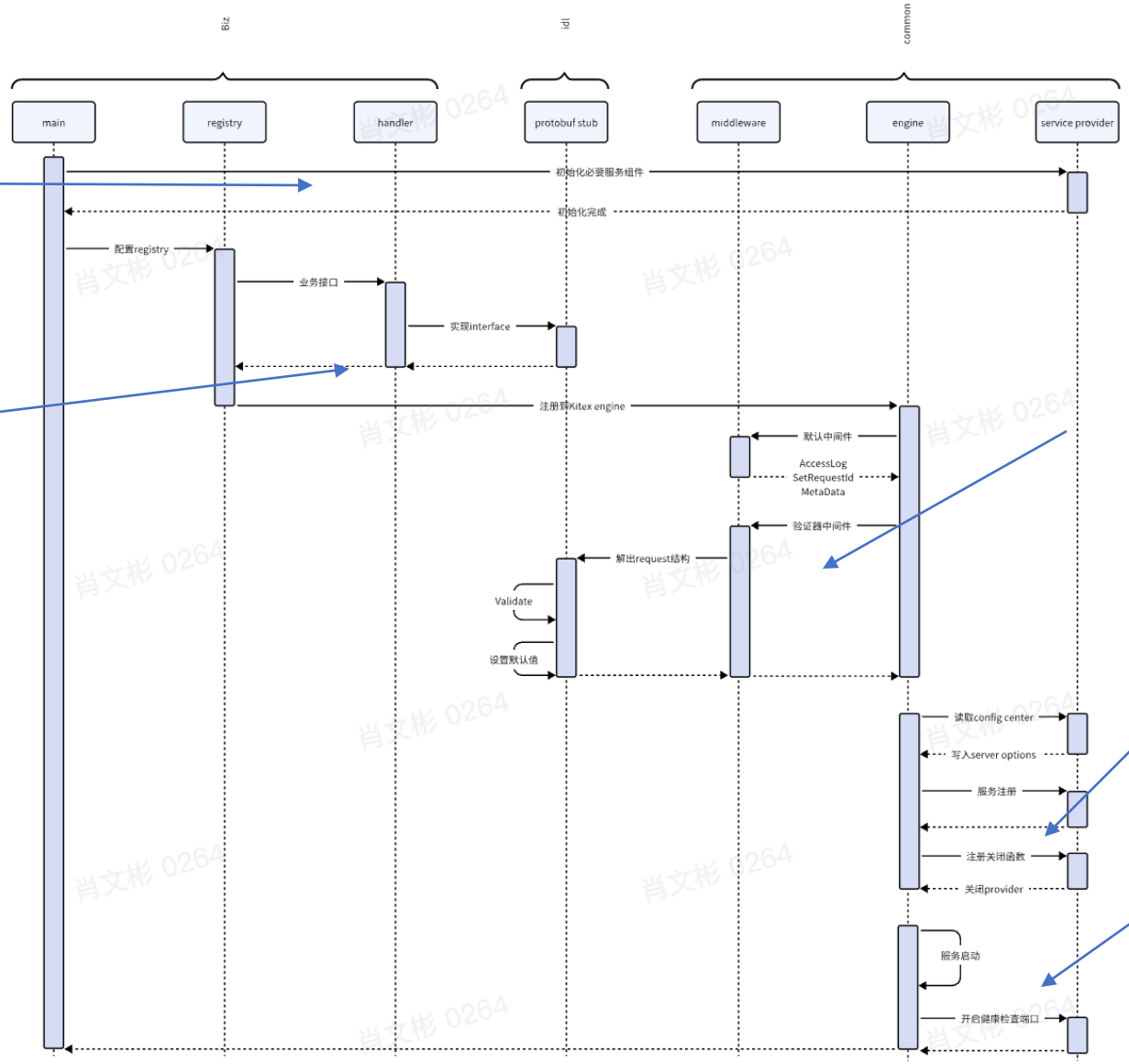
4. 设置服务启动选项参数

5. 服务启动

Kitex启动

1. 初始化服务组件

2. 根据IDL实现业务接口
配置registry并注册



3. 加入默认前置中间件
验证入参，设定默认值

4. 设置服务启动选项参数

5. 服务启动

03

可观测性集成

框架与三大指标的结合

日志

支持常用主流库

支持zap、logrus、slog、fmt，同时实现了hlog、klog的interface

支持field的json格式

不同的输出模式

既可以stdout到控制台，也可以日志切割到本地文件

接入trace和sentry

作为event附在span中，将traceId输出在log中。同时也上报sentry

灵活的上下文参数

预设key值，打印日志时如果ctx中有预设的key，那么会自动打印value

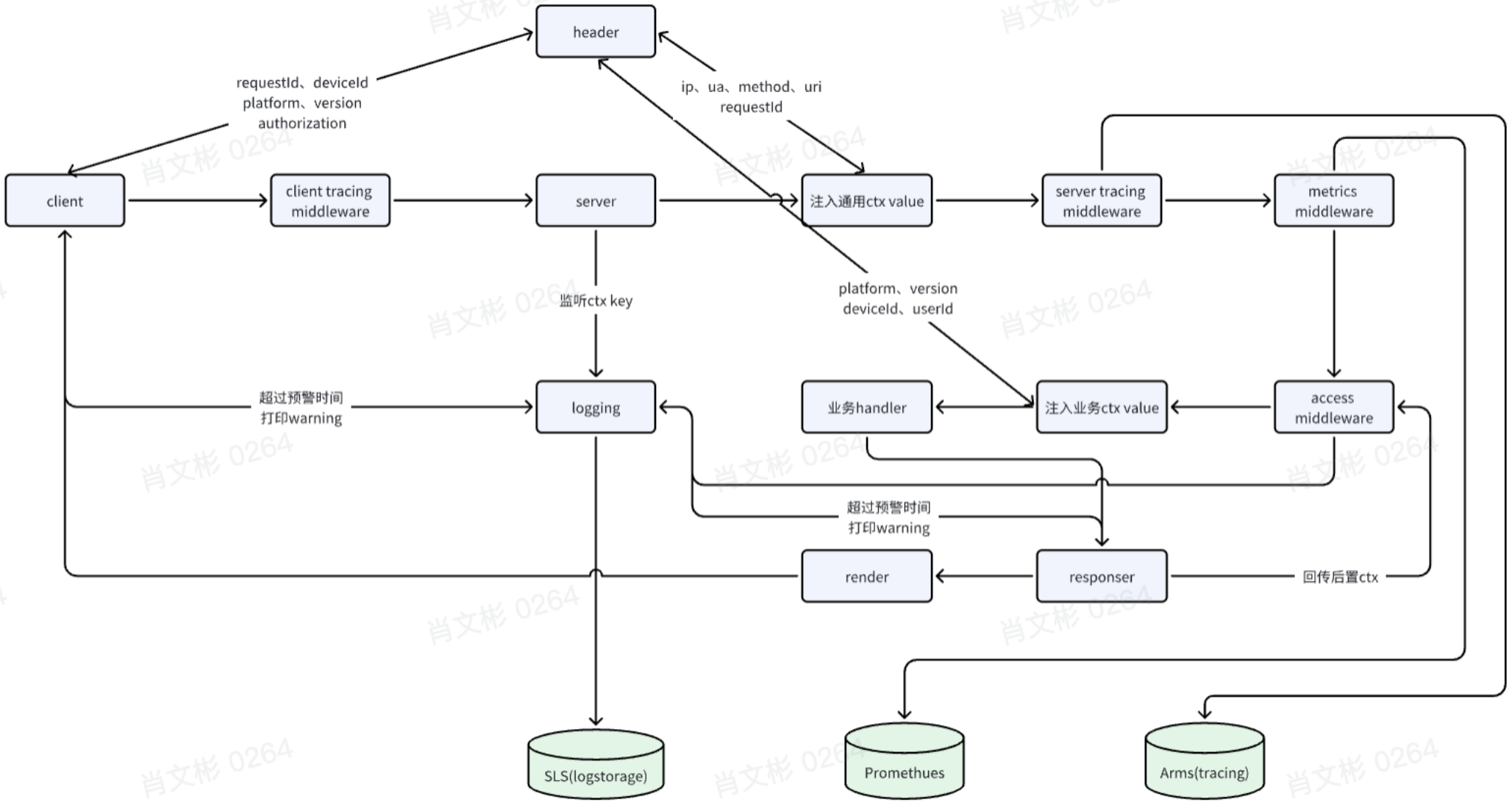
非常重要的功能，是很多打印元数据的基础

AccessLog

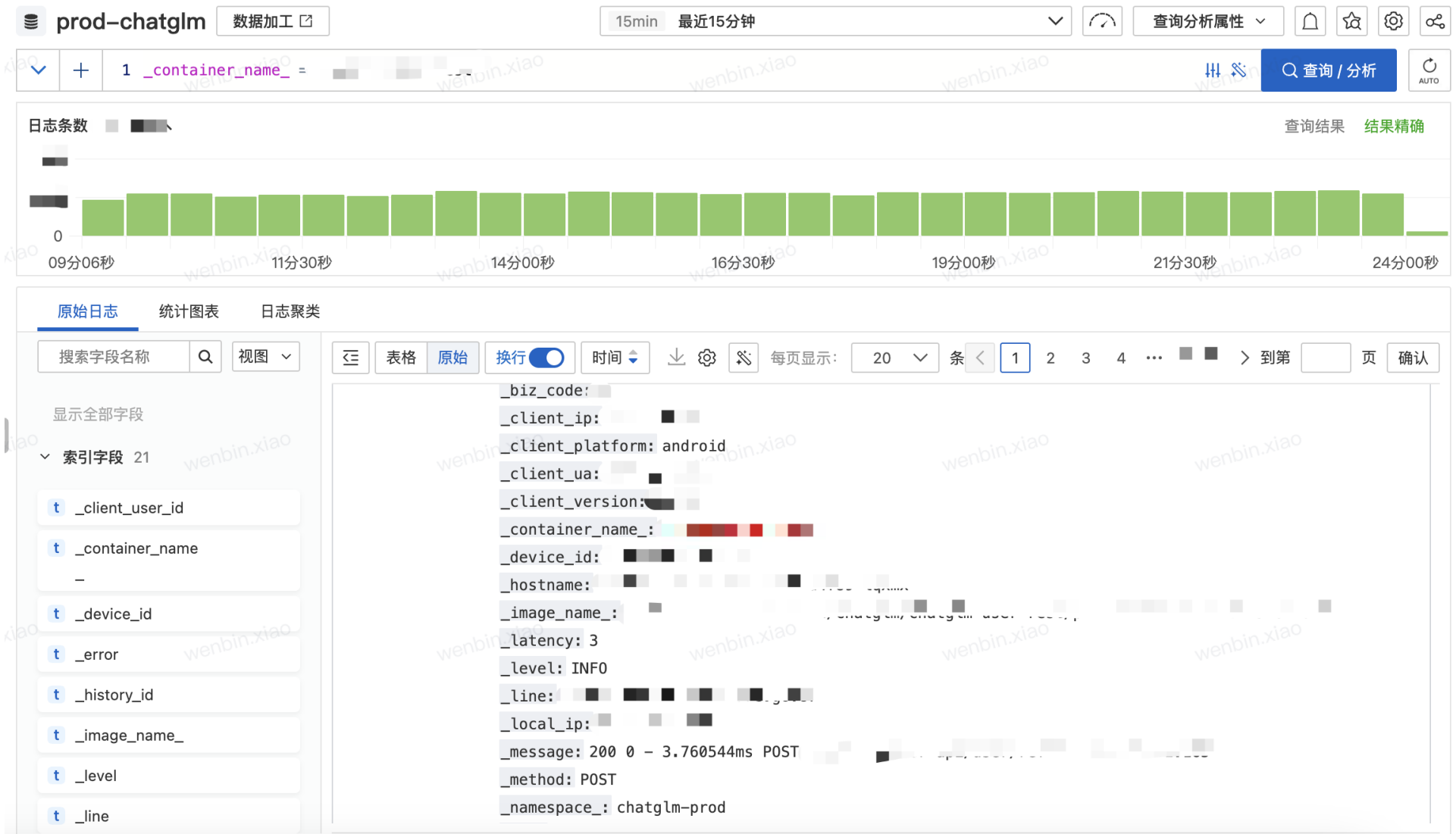
```go
func (h *Middleware) AccessLog(ctx context.Context, c *app.RequestContext) {  1 个用法  xiaowenbin
    now := time.Now()
    c.Next(ctx)
    if bc, ok := c.Get(backwardCtxKey); ok {
        ctx = bc.(context.Context)
    }
    statusCode, bizCode, latency := c.Response.StatusCode(), c.GetInt(bizCodeKey), time.Since(now)
    msg := fmt.Sprintf(
        format: "%d %d - %s %s %s %s %s",
        statusCode,
        bizCode,
        latency.String(),
        c.Method(),
        c.Request.Path(),
        c.ClientIP(),
        c.GetHeader( key: "Referer"),
    )
    lf := map[string]any{
        "provider":    "access_log",
        "status_code": statusCode,
        "biz_code":    bizCode,
        "latency":     latency.Milliseconds(),
    }
    logging.WithFields(lf).CtxInfo(ctx, msg)
}

func (h *Middleware) SetClientMetadata(ctx context.Context, c *app.RequestContext) {  1 个用法  新 *
    ctx = context.WithValue(ctx, constants.CtxClientIpKey, c.ClientIP())
    ctx = context.WithValue(ctx, constants.CtxMethodKey, string(c.Method()))
    ctx = context.WithValue(ctx, constants.CtxUriKey, string(c.Request.RequestURI()))
    ctx = context.WithValue(ctx, constants.CtxClientUaKey, string(c.UserAgent()))
    origin, referer := string(c.GetHeader( key: "Origin")), string(c.GetHeader( key: "Referer"))
    if !utils.IsEmptyStr(referer) {
        ctx = context.WithValue(ctx, constants.CtxRefererKey, referer)
    }
    if utils.IsEmptyStr(origin) && strings.HasPrefix(referer,  prefix: "http") {
        u, _ := url.Parse(referer)
        origin = u.Scheme + "://" + u.Host
    }
    if !utils.IsEmptyStr(origin) {
        ctx = context.WithValue(ctx, constants.CtxOriginKey, origin)
    }
    c.Next(ctx)
}
```

```go
func (h *BaseHandler) DefaultResponser(ctx context.Context, c *app.RequestContext, o constants.IRestOutput) {  1 个用
            http.StatusUnauthorized,
            http.StatusForbidden,
            http.StatusNotFound,
            http.StatusMethodNotAllowed,
            http.StatusInternalServerError,
            http.StatusTooManyRequests,
        }, bizCode) {
            statusCode = bizCode
        } else {
            statusCode = http.StatusOK
        }
    }
    if rid := ctx.Value(constants.CtxRequestIdKey); rid != nil {
        o.SetRequestId(rid.(string))
    }
    var latency time.Duration
    if requestTime := ctx.Value(constants.CtxRequestTimeKey); requestTime != nil {
        latency = time.Since(requestTime.(time.Time))
    }
    lf["status_code"], lf["biz_code"], lf["latency"] = statusCode, bizCode, latency.Milliseconds()
    if data != nil {
        lf["response_data"] = data
    }
    c.Set(backwardCtxKey, ctx)
    c.Set(bizCodeKey, bizCode)
    l := logging.WithContext(ctx).WithFields(lf)
    if utils.HasErr(err) {
        if statusCode == http.StatusUnauthorized || msg == responseBadRequest.Message {
            l = l.WithField( key: "authorization", string(c.GetHeader(constants.HeaderAuthorizationKey)))
        }
        if warningHttpCodes[statusCode] {
            l.Warning( msg: "rest response %d: %s", statusCode, err.Error())
        } else if errorx.IsCustomInfo(err) {
            l.WithField( key: "custom_message", err.Error()).Warning( msg: "rest response custom message")
        } else {
            l.Error(err,  msg: "rest response err: %s", msg)
        }
    } else {
        l.Debug( msg: "rest response info")
    }
    warningLatency := conf.Timeout.Warning
    if utils.IsPositiveNumber(warningLatency) && latency > warningLatency && !IsAllowedPath(c, ignoreSlowWarningPath
        l.Warning( msg: "rest response too slow")
    }
    if string(c.Response.Header.ContentType()) == ContentTypeStream {
        c.AbortWithStatus(statusCode)
    } else {
        c.AbortWithStatusJSON(statusCode, o)
    }
```

Hertz日志处理


header
requestId、deviceId
platform、version
authorization
ip、ua、method、uri
requestId
client
client tracing middleware
server
注入通用ctx value
server tracing middleware
metrics middleware
监听ctx key
platform、version
deviceId、userId
超过预警时间
打印warning
logging
业务handler
注入业务ctx value
access middleware
超过预警时间
打印warning
render
responser
回传后置ctx
SLS(logstorage)
Promethues
Arms(tracing)

日志


prod-chatglm
数据加工
15min 最近15分钟
查询分析属性
1 _container_name_ =
查询 / 分析
AUTO
日志条数
查询结果
结果精确
0
09分06秒
11分30秒
14分00秒
16分30秒
19分00秒
21分30秒
24分00秒
原始日志
统计图表
日志聚类
搜索字段名称
视图
显示全部字段
索引字段 21
_client_user_id
_container_name_
_device_id
_error
_history_id
_image_name_
_level
_line
表格
原始
换行
时间
每页显示:
20
条
1
2
3
4
到第
页
确认
_biz_code:
_client_ip:
_client_platform: android
_client_ua:
_client_version:
_container_name_:
_device_id:
_hostname:
_image_name_:
_latency: 3
_level: INFO
_line:
_local_ip:
_message: 200 0 - 3.760544ms POST
_method: POST
namespace: chatglm-prod

指标

使用社区中间件

- 社区提供的开箱即用prometheus中间件基本满足我们的使用需求

  Hertz参考github.com/hertz-contrib/monitor-prometheus

  Kitex参考github.com/kitex-contrib/monitor-prometheus
- 指标默认带有qps、时延、运行时等常规指标
- 我们引入一个额外的包提供promreg的句柄，在启动时注入到中间件暴露的WithRegistry方法中
- 业务直接使用promreg上报自定义指标
- 统计性能指标：各路由-状态码QPS、各路由时延、内存、协程数等

链路追踪

遵循OpenTelemetry协议

通用的的provider

生成traceId

实现IDGenerator接口，优先将ctx的requestId作为traceId

如果不存在或不合标准，用默认的随机规则

链路传播

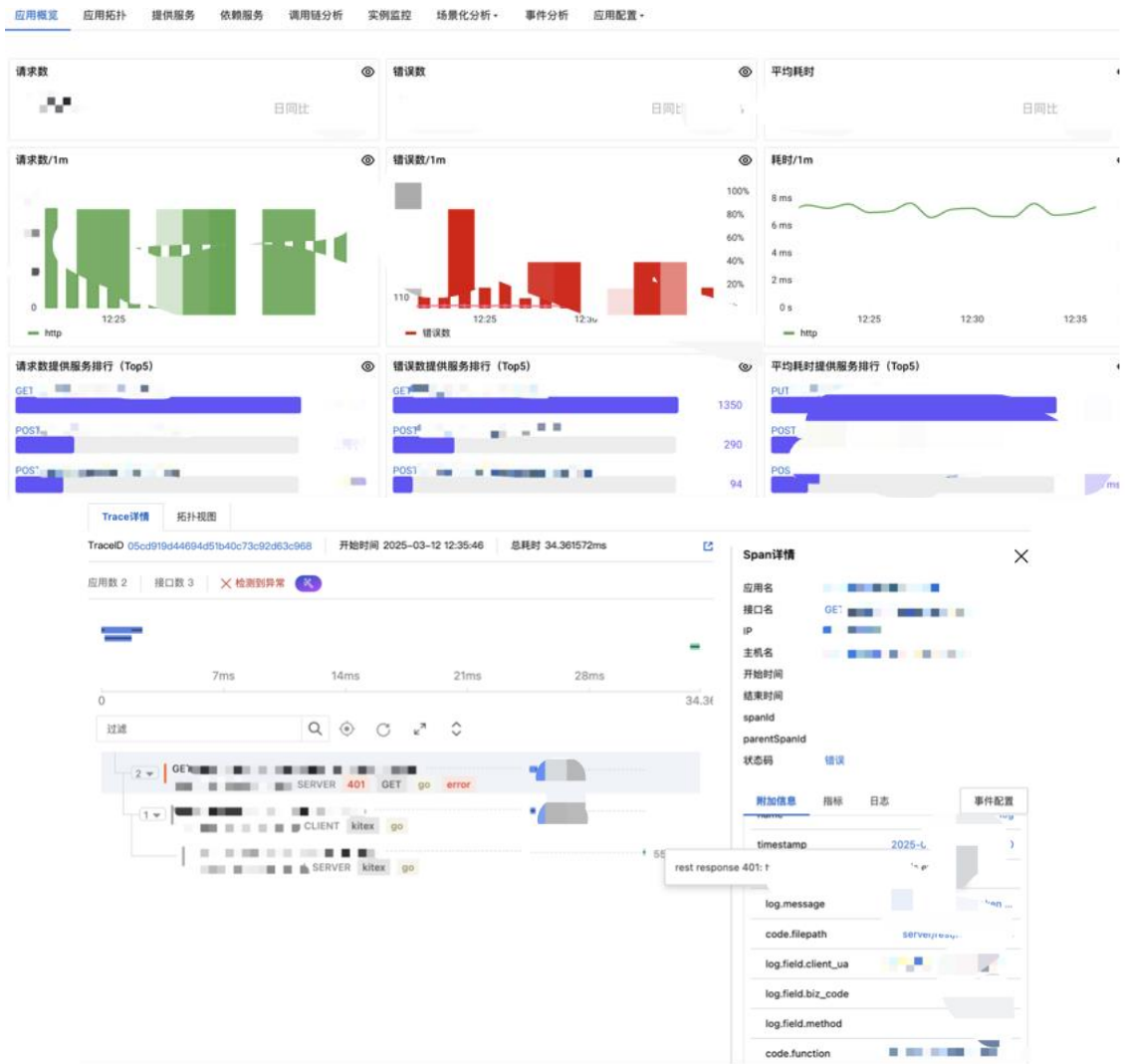
默认使用TraceContext、Baggage进行传播

全链路追踪

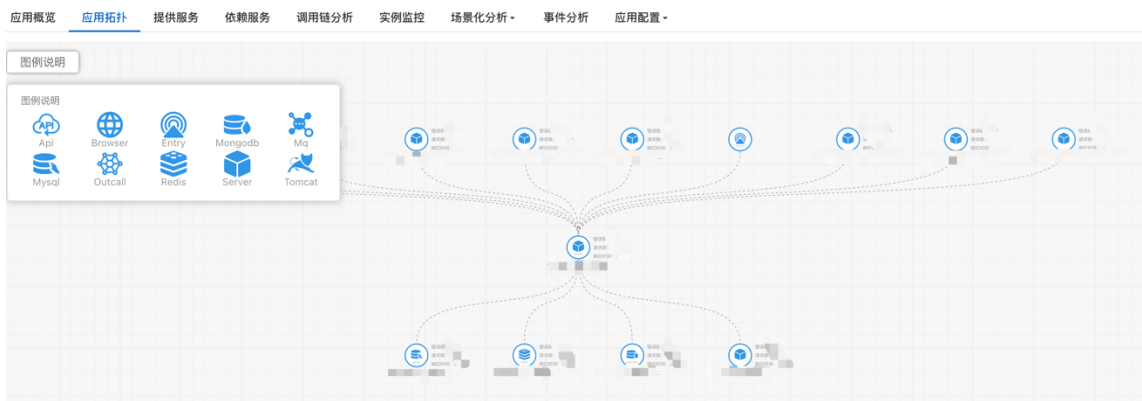
包括不限于Rest/Rpc的server/client、MySQL、Mongo、Redis、ES等

- Hertz参考社区组件github.com/hertz-contrib/obs-opentelemetry
- Kitex参考社区组件github.com/kitex-contrib/obs-opentelemetry
- 各个常用组件库都会提供hook，参考https://opentelemetry.io/ecosystem/registry/?language=g

链路追踪


应用概览
应用拓扑
提供服务
依赖服务
调用链分析
实例监控
场景化分析
事件分析
应用配置
请求数
错误数
平均耗时
请求数/1m
错误数/1m
耗时/1m
请求数提供服务排行（Top5）
错误数提供服务排行（Top5）
平均耗时提供服务排行（Top5）
Trace详情
拓扑视图
Span详情


应用概览
应用拓扑
提供服务
依赖服务
调用链分析
实例监控
场景化分析
事件分析
应用配置
快捷筛选
调用次数
Span列表
Trace列表
慢Trace分析
错Trace分析


应用概览
应用拓扑
提供服务
依赖服务
调用链分析
实例监控
场景化分析
事件分析
应用配置
图例说明
Api
Browser
Entry
Mongodb
Mq
Mysql
Outcall
Redis
Server
Tomcat

问题告警

基于日志错误量

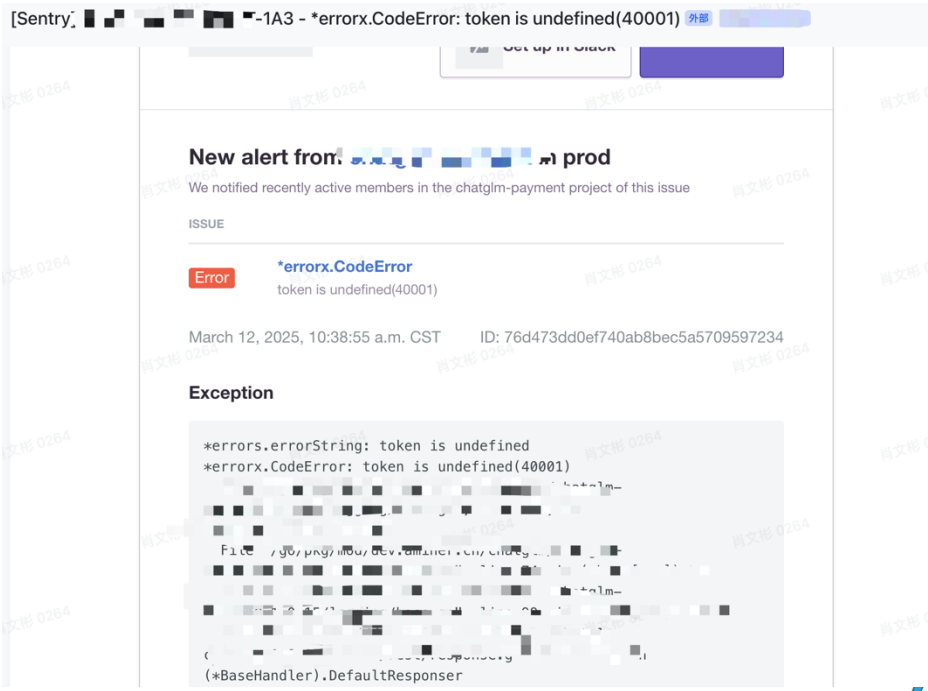
偏重于错误量环比上升，可编写规则

飞书机器人形式

基于Sentry

偏重于错误明细，可定位堆栈，各错误占比

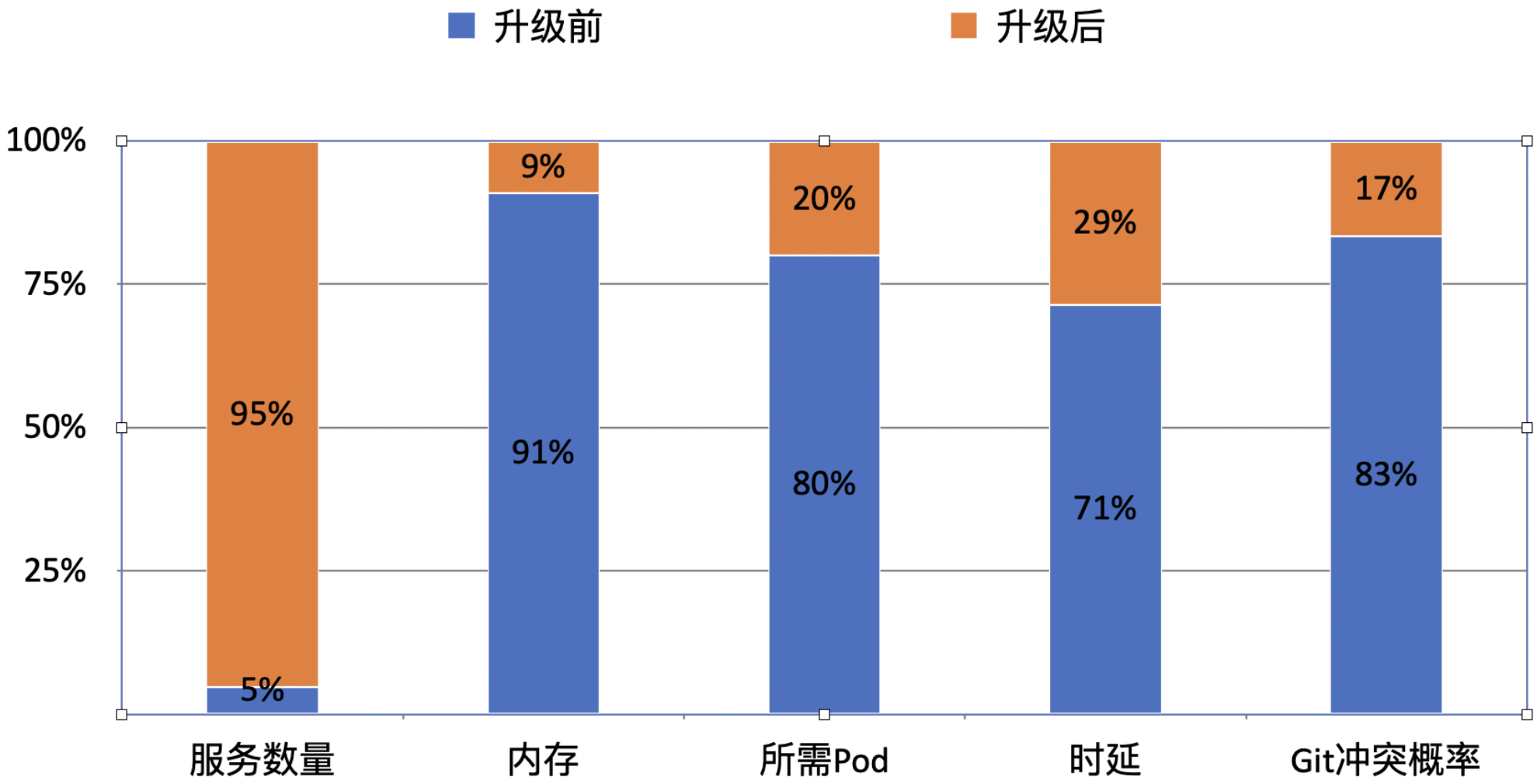
邮件形式



04 总结与展望

升级的收益与框架feature

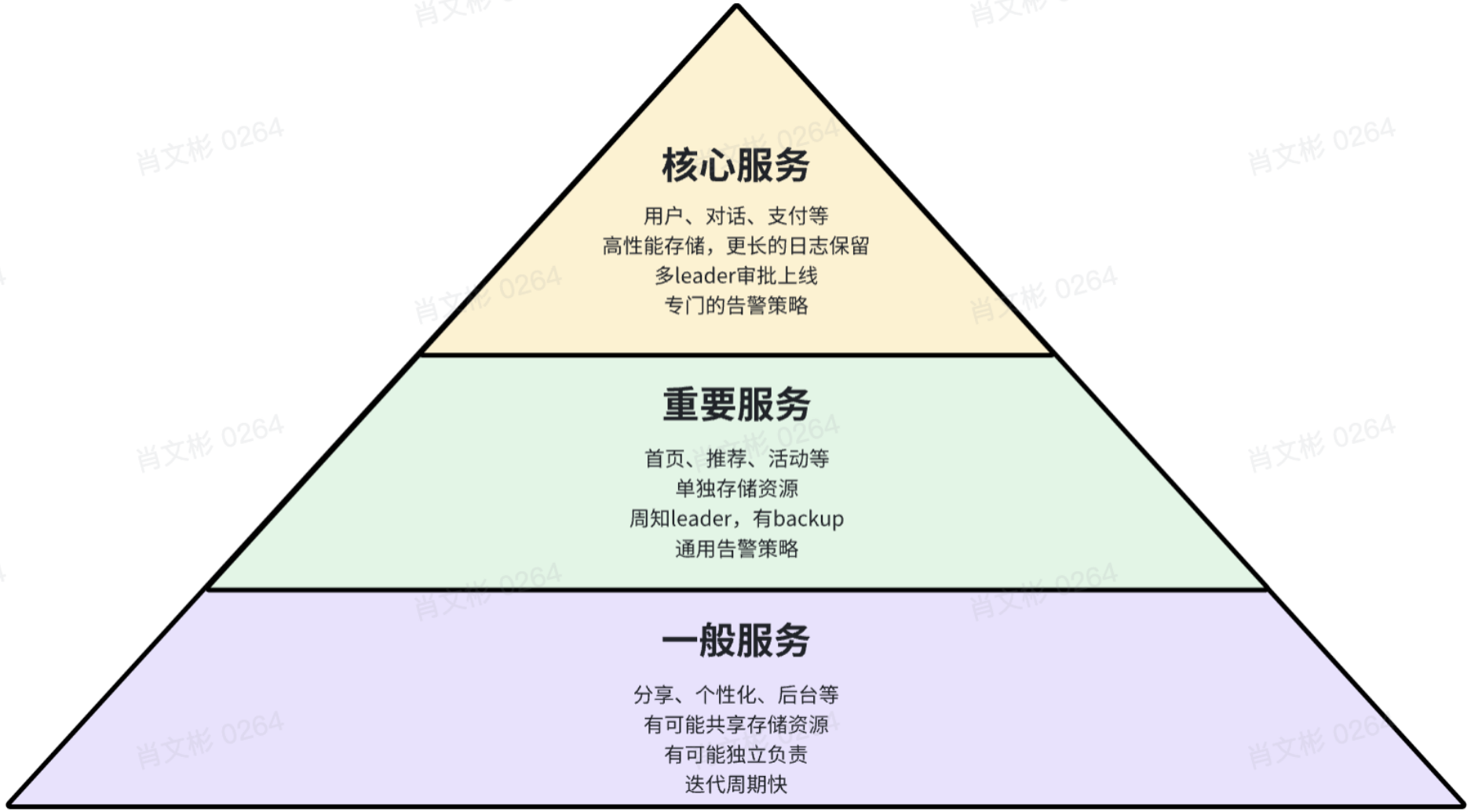
升级结果



注：内存与Pod按对话服务统计，时延按高频接口统计

整体硬件资源约节省90%

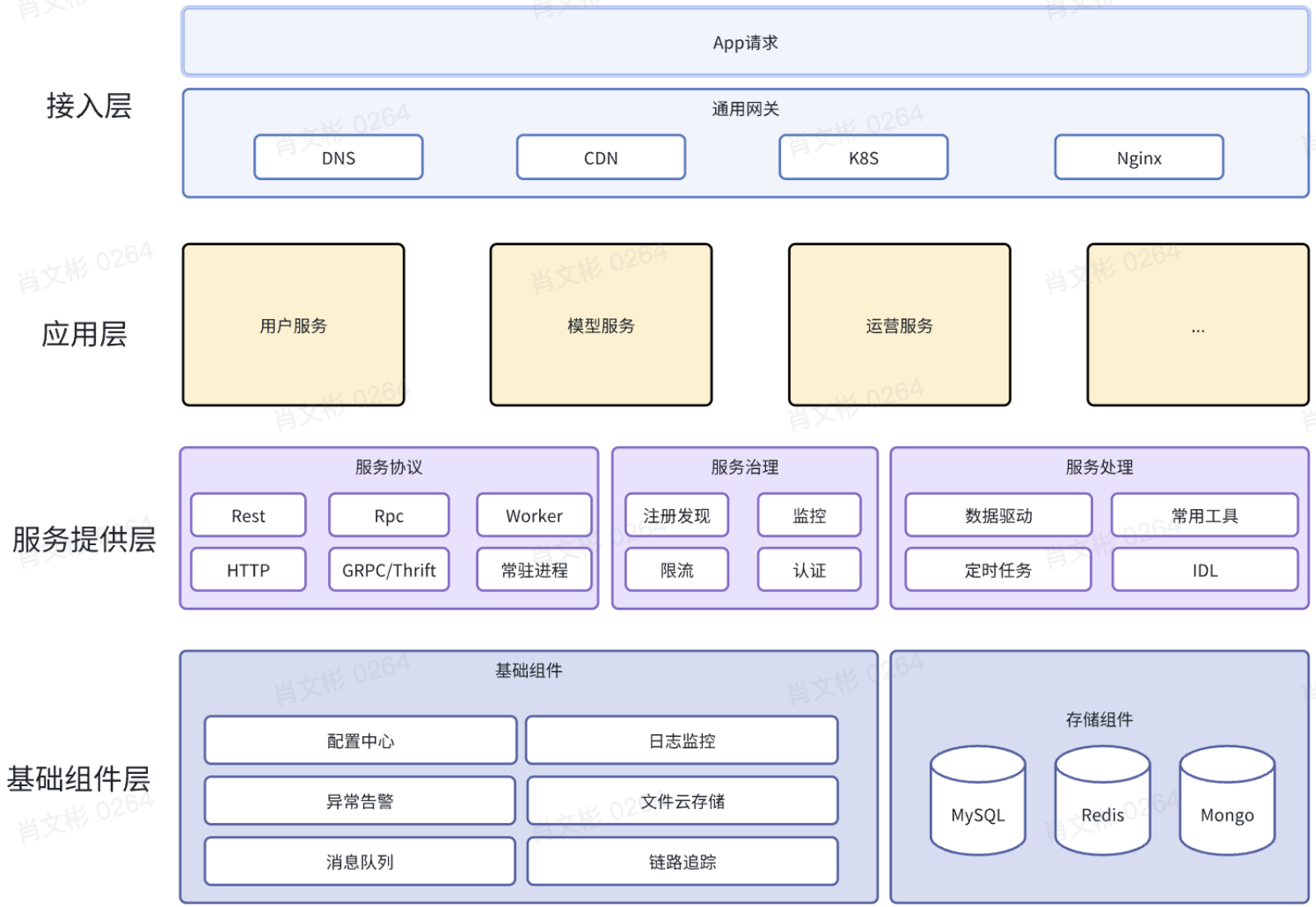
服务分级



其他收益

- 更新SDK即可体验feature
- 众多容错机制，性能优化，友好封装，即使刚上手的同学也可保障服务下限
- 仅需配置中心的参数，无需繁杂的环境变量，拉下代码仓库就可轻松启动服务
- 常用工具函数与组件调用，减少重复轮子
- 预设默认门面方法，同时提供可扩展接口，业务可实现自定义插件，提高扩展性
- 退出服务自动回收Provider资源
- 统一格式的日志、部署流程，大大利好运维
- Python需要轮询配置中心获取最新配置，开启多个worker会成倍增加压力，而Golang使用长连接监听，被动更新配置，有更好实时性的同时减少了请求频率

系统架构图


接入层
App请求
通用网关
DNS
CDN
K8S
Nginx
应用层
用户服务
模型服务
运营服务
...
服务提供层
服务协议
Rest
Rpc
Worker
HTTP
GRPC/Thrift
常驻进程
服务治理
注册发现
监控
限流
认证
服务处理
数据驱动
常用工具
定时任务
IDL
基础组件层
基础组件
配置中心
日志监控
异常告警
文件云存储
消息队列
链路追踪
存储组件
MySQL
Redis
Mongo

框架feature

SSE/Websocket应用

流式与长连接的接口，目前也在尝试应用中

服务中心的更多应用

目前仅进行注册，实际更多调用k8s域名

各服务可感知彼此，尝试对metadata的应用

提高gRPC、proto兼容性

Python通过gRPC调用Kitex还存在一些影响不大的细节兼容性问题

进一步排查底层原因后反馈社区

回馈社区

自身运行比较稳定的feature，可通过PR提供插件

THANKS

CloudWeGo