字节跳动微服务合并编译实践

尹旭然

字节跳动后端研发工程师

InfoQ 极客传媒

已结束 北京站 2000 人

QCon 全球软件开发大会 暨智能软件开发生态展

架构演进 | AIGC
研发效能 | 向量数据库

6月14-15日 深圳站 1000 人

ArchSummit 全球架构师峰会 暨智能软件开发生态展

业务架构 | 高可用
AI 架构 | 架构师成长

10月18-19日 上海站 1500 人

QCon 全球软件开发大会 暨智能软件开发生态展

AIGC | AI 原生应用
软件工程 | 金融科技

4月 5月 6月 8月 10月 12月

5月17-18日 北京站 2000 人

AiCon 全球人工智能开发与应用大会 暨大模型应用生态展

通用大模型 | AI Agent
LLMOps | AI Infra

8月16-17日 上海站 1000 人

FCon 全球金融科技大会 暨智能软件开发生态展

金融大模型 | 核心系统
数字化转型 | 数据安全

12月20-21日 北京站 1000 人

ArchSummit 全球架构师峰会 暨智能软件开发生态展

未来架构 | 高可用高并发
AI 应用 | 架构师思维

极客邦科技 2024 年会议规划

促进软件开发及相关领域知识与创新的传播

访问大会官网

参会咨询

目录

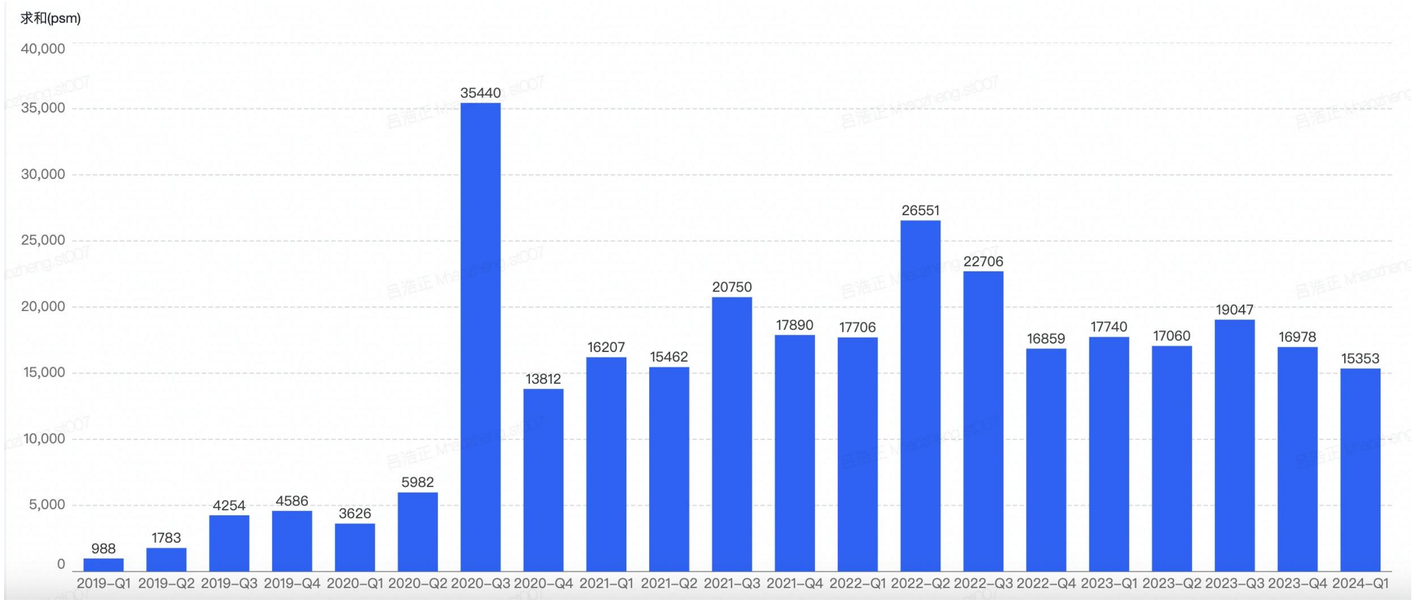
字节跳动微服务的背景

- 字节跳动微服务的现状 – 微服务过微
- 微服务过微的原因
- 微服务过微的危害

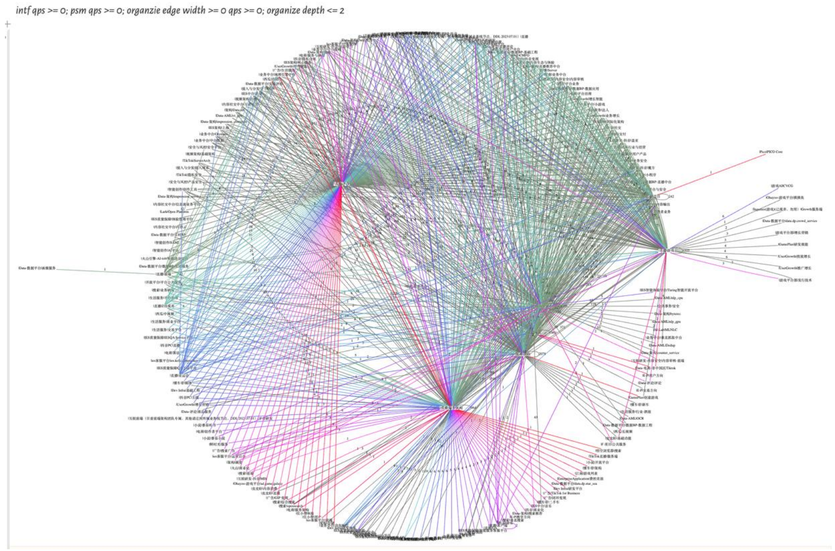
字节跳动微服务的现状 - 微服务过微

字节跳动微服务过微的现状：
- 微服务数量超过30w，部分业务人均维护微服务数量超过10个
- 微服务的数量仍然在快速的增长当中
- 微服务的调用关系错综复杂

导致该现状的原因：
- 自18年以来，业务高速增长，研发效率优先
- 缺乏统一的管控措施和规范，微服务拆分粒度和把控不合理



按季度新增微服务数量



某业务线微服务调用拓扑图

微服务过微的危害

运行效率

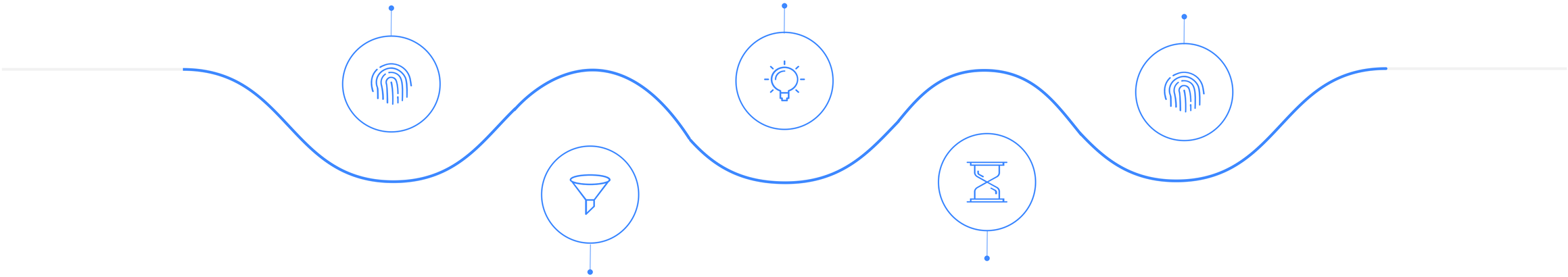
序列化、服务治理开销大，消耗资源过多；链路过长导致时延过高

稳定性

链路过长导致所需准备的容灾预案工作难度提升，下游任何一跳的服务故障都可能导致整个链路受损

架构观测

链路庞大导致全链路评估和观测的成本高



研发效率

人均维护的服务数量过多导致研发效率下降

服务治理

依赖的服务过多导致超时、限流等配置增多，合理配置的难度大幅上升

如何解决微服务过微的问题

- 如何解决微服务过微的问题
- 合并编译的方案是什么
- 合并编译的优势和挑战

如何解决微服务过微的问题

新增治理

- 服务增量管控

存量治理

- 服务治理
- 链路治理
- 领域治理

能删就删

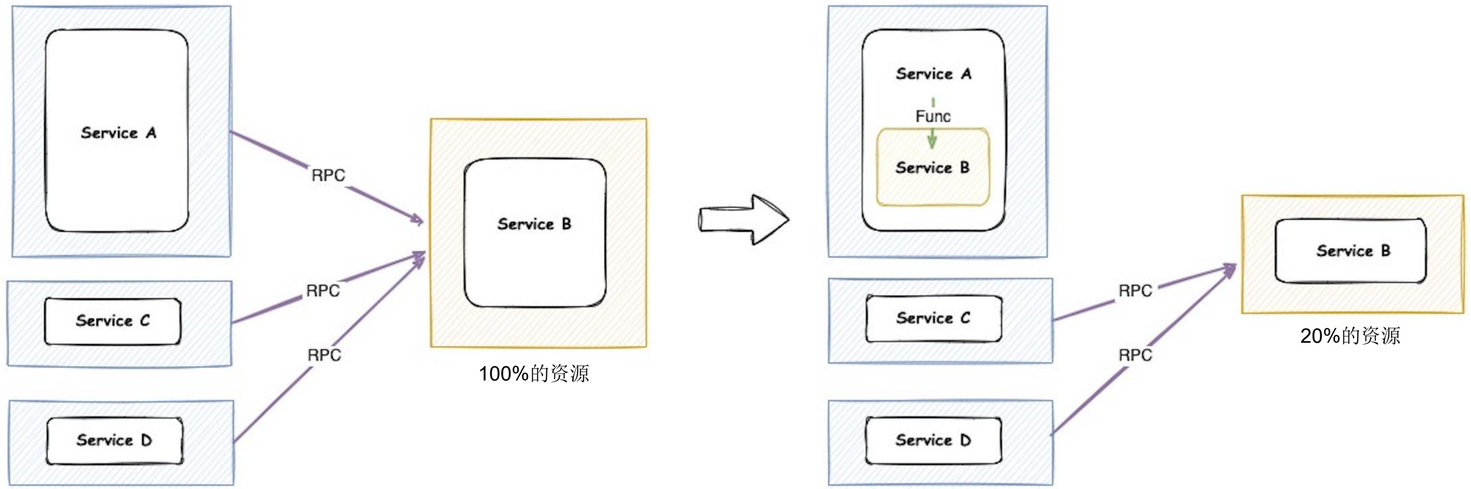
- 过微服务
- 长尾服务
- 重复定位服务

能合就合

- 微服务 SDK 化改造
- 合并编译

合并编译是什么

合并编译：两个（或多个）微服务（独立的多个进程），在**编译期间合并**为一个二进制，以**一个进程**的方式运行。



流量比例 A : C : D = 8 : 1 :1 时的合并示意图

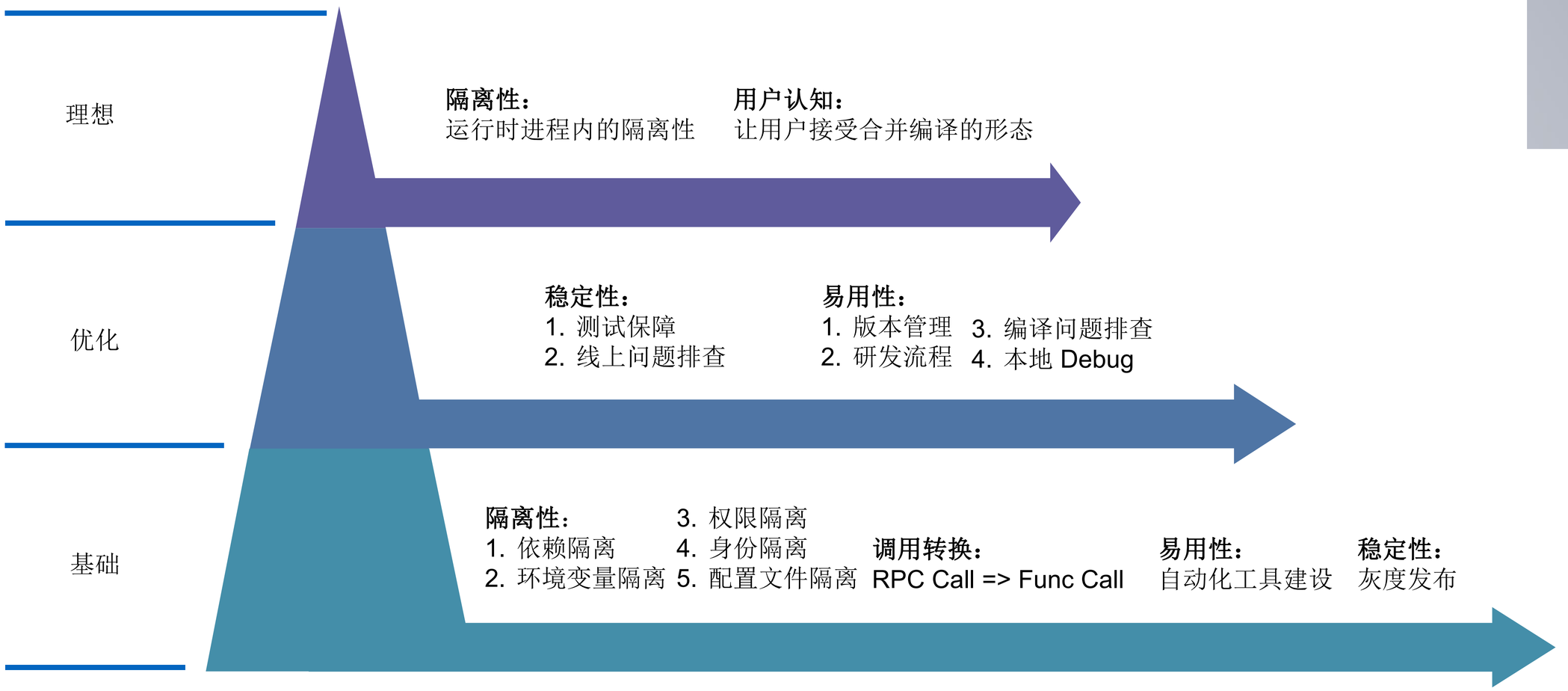
优势

- 极致的性能
- 灵活的合并与拆分
- 稳定性提高
- 链路复杂度

劣势

- 进程内隔离性
- 版本管理

合并编译面临的挑战


理想
优化
基础
隔离性：
运行时进程内的隔离性
用户认知：
让用户接受合并编译的形态
稳定性：
1. 测试保障
2. 线上问题排查
易用性：
1. 版本管理
2. 研发流程
3. 编译问题排查
4. 本地 Debug
隔离性：
1. 依赖隔离
2. 环境变量隔离
3. 权限隔离
4. 身份隔离
5. 配置文件隔离
调用转换：
RPC Call => Func Call
易用性：
自动化工具建设
稳定性：
灰度发布

合并编译如何解决面临的挑战

- 依赖隔离
- 环境变量隔离
- 调用转换
- 灰度上线
- 版本管理

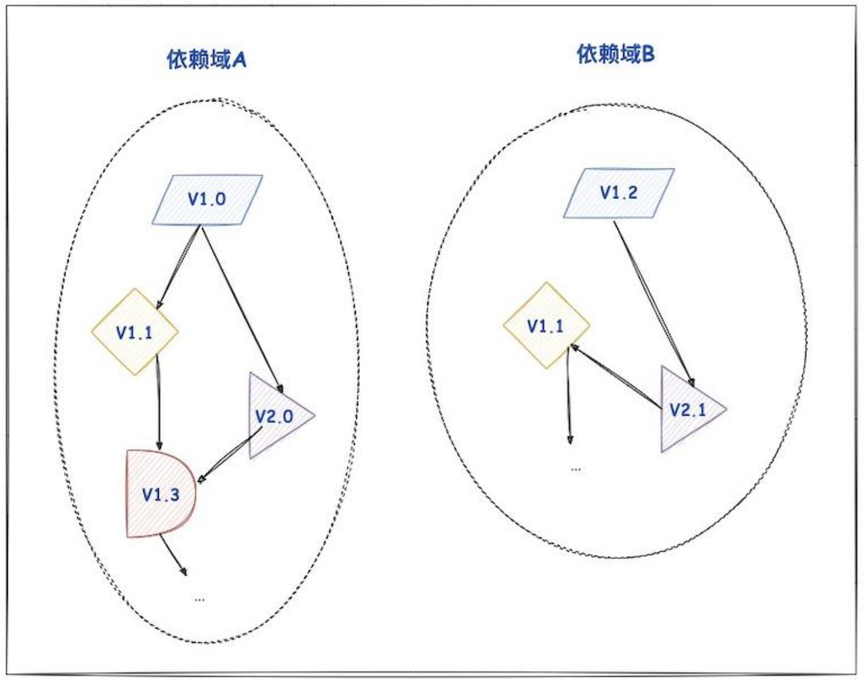
合并编译技术介绍 - 依赖隔离

依赖版本冲突：
1. 依赖版本不兼容

无法收敛：
每次合并都有新的 case 出现

全局单例 SDK 冲突：
1. 初始化
2. 配置合并
3. 相互影响

业务有感



合并编译技术介绍 - 依赖隔离

```
package main

import (
    "fmt"

    "github.com/cloudwego/kitex"
)
```

go mod 管理方式：

1.标准库 fmt：$GOROOT/src/fmt

2.三方库 kitex：$GOPATH/pkg/mod/github.com/…/kitex

3.replace：将依赖库的版本替换为其他版本或修改依赖库的路径

给三方库加上不同的 namespace 并且修改三方库的 import 路径呢

合并编译技术介绍 - 依赖隔离

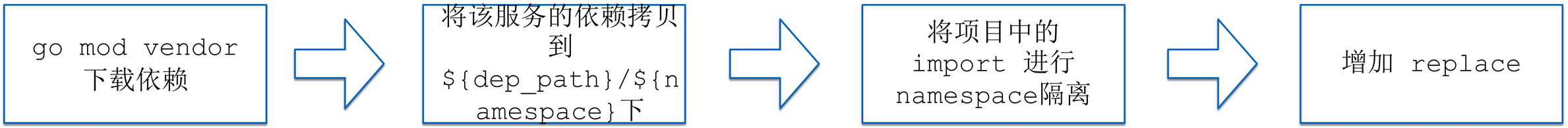
```
package main

import (
    "svc_inline/namespacea/github.com/cloudwego/kitex"

    "svc_inline/namespaceb/github.com/cloudwego/kitex"
)
```

```
/tmp
└── svc_inline
    ├── namespacea
    │   └── github.com
    │       └── cloudwego
    │           └── kitex
    └── namespaceb
        └── github.com
            └── cloudwego
                └── kitex
```

```
replace svc_inline/namespacea/.../kitex => /tmp/svc_inline/namespacea/…/kitex
replace svc_inline/namespaceb/.../kitex => /tmp/svc_inline/namespaceb/…/kitex
```

```
replace svc_inline/=> /tmp/svc_inline
```

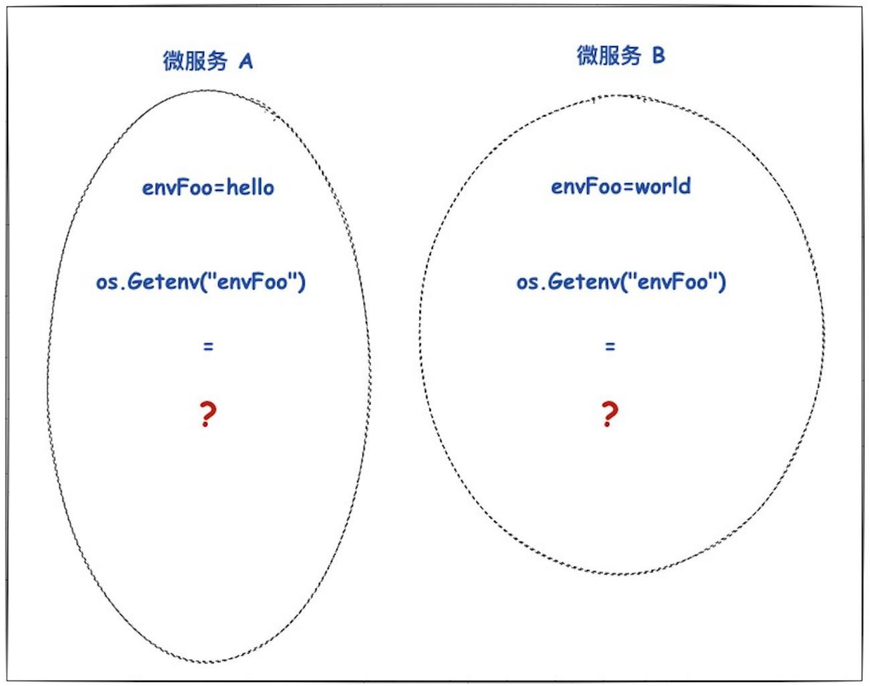
合并编译技术介绍 - 依赖隔离



合并编译技术介绍 - 环境变量隔离

Golang 获取环境变量

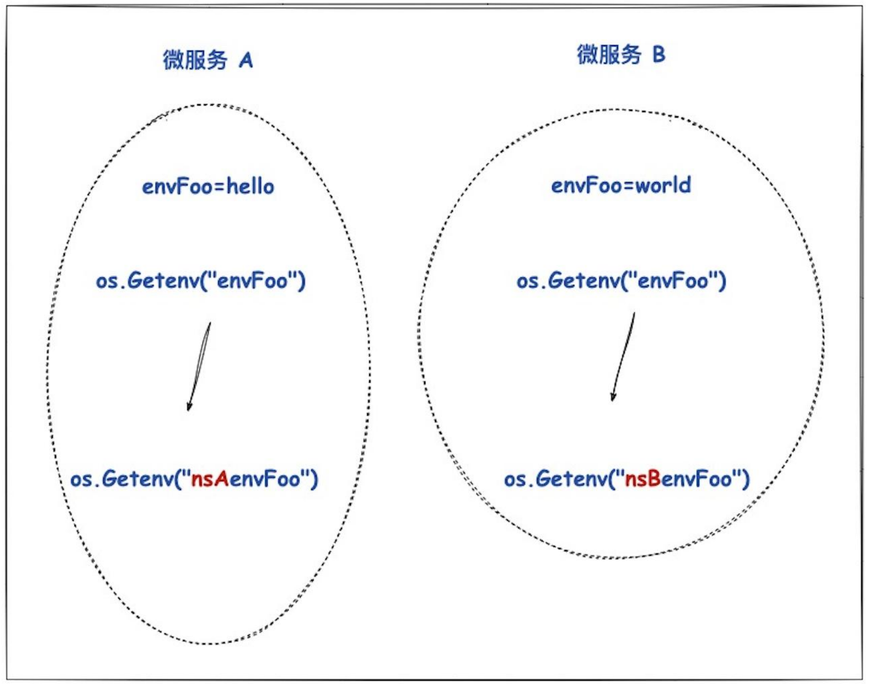
```
func Getenv(key string) string {
    testlog.Getenv(key)
    v, _ := syscall.Getenv(key)
    return v
}

func Setenv(key, value string) error {
    err := syscall.Setenv(key, value)
    if err != nil {
        return NewSyscallError("setenv", err)
    }
    return nil
}
```



合并编译技术介绍 - 环境变量隔离

获取环境变量也做隔离

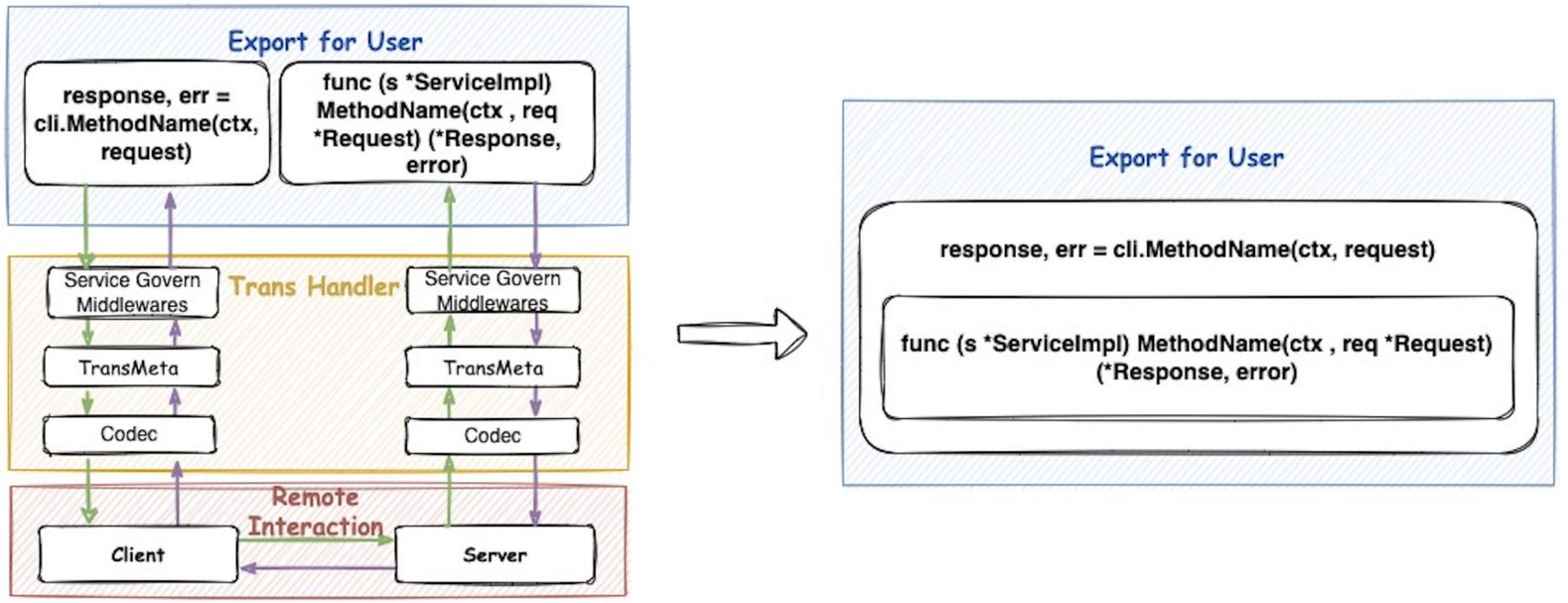
```
func Getenv(key string) string {
    testlog.Getenv(key)
    v, _ := syscall.Getenv("nsA" + key)
    return v
}

func Setenv(key, value string) error {
    err := syscall.Setenv("nsA" + key, value)
    if err != nil {
        return NewSyscallError("setenv", err)
    }
    return nil
}
```



合并编译技术介绍 - 调用转换

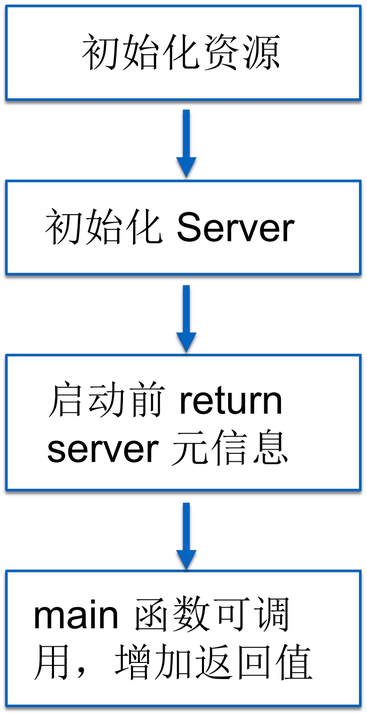
如何实现 Function Call：

1. 获得 Server 对外暴露的接口
2. 将该接口的实现注入到 Client 中去



合并编译技术介绍 - 调用转换

获得 Server 对外暴露的接口




```go
func main() {
    // init logic
    mysql.Init()
    mq.Init()
    // init server
    svr := kitex_c.NewServer(handler)
    // start server
    svr.Run()
}
```

```go
func Main() kitex.ServerInfo {
    // init logic
    mysql.Init()
    mq.Init()
    // init server
    svr := kitex_c.NewServer(handler)
    return svr.ServerInfo()
    // start server
    svr.Run()
}
```

合并编译技术介绍 - 调用转换

将 Server 接口的实现注入到 Client 中去

Kitex Client 定义

```
type Client interface {
    Call(ctx context.Context, method string, request, response interface{}) error
}
```

kitex_gen 中 Client 初始化

```
kc, err := client.NewClient(serviceInfo(), options...)
```

kitex_gen 中 Client 初始化

```
kc, err := client.NewServiceInlineClient(serviceInfo(), serverInfo ,options…)
```

```
server.Main()
```

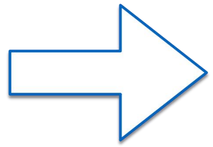
ser

以上实现在 cloudwego/kitex 中

合并编译技术介绍 - 灰度上线

一个理想的灰度上线是什么样子的呢 🤔

1. 灰度期间保证一套代码
2. 灰度流量的比例能够自由确定
3. 能够按需求放量，比如方法级别放量
4. 出现问题能够及时的回滚
5. 能够在灰度期间看出收益



1. 一个镜像里面有两个二进制：原始 & 合并编译，通过环境变量控制
2. 通过配置（静态或动态）控制合并编译的流量比例

灰度方式	如何操作	优势
普通需求上线	-	上线迅速，能够快速观察到效果；监控、报警不需要额外配置
同名集群放量	新建一个同名集群，通过增加实例数量控制放量比例	不同阶段可停留不同的时间，有较为充分的时间观察指标
开实验放量	新建集群通过开 AB 实验的方式放量	可以观测到业务指标的变化
通过配置放量	和实现强相关	放量灵活，比如可以按照不同的方法进行放量

合并编译技术介绍 - 灰度上线

自定义是否走合并编译逻辑：

1. 框架注入两个 Client，根据 UseServiceInline 函数返回值决定用哪个 Client
2. 用户自行实现 UseServiceInline 函数，这个函数的实现是一个文件

```
func clientCallMethod(ctx, methodName, req) {
    if UseServiceInline(cdx, method, req) {
        // 走合并编译
        serviceInlineClient.Call(ctx, methodName, req, resp)
    } else {
        // 调用下游
        normalClient(ctx, methodName, req, resp)
    }
}
```

框架侧注入实现

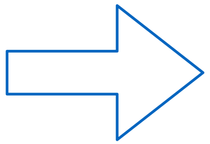
```
package condition
import (
        xxx
    )


 func UseServiceInline(ctx, name, req)  bool {
      // xxx
 }
```

用户自定义逻辑

合并编译技术介绍 - 版本管理

版本管理有哪些痛点？

1. 版本的不一致
2. 不知道版本分布
3. 版本选择复杂



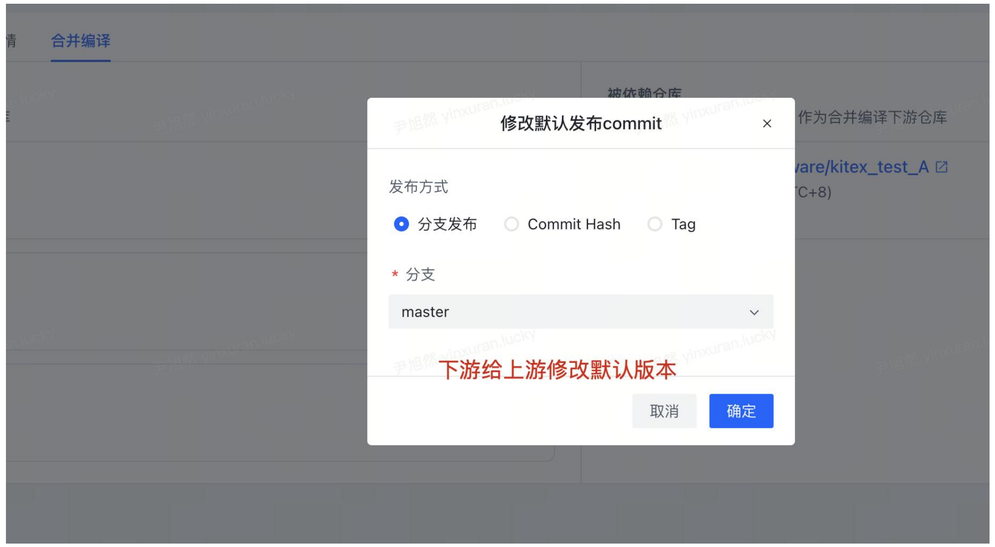
1. 平台提供基础能力
2. 用户学习一种新的上线方式

类型	区别	优势	适用场景
最终一致性	版本能够在一段时间后收敛	有较为充分的灰度时间，没问题了再上线	1. 要求稳定性的场景 2. 中台化场景 3. 业务线的基础服务
强一致性	版本在较短时间内收敛	版本能够立即生效	1. 上下游密切相关的服务 2. 下游只给上游调用的服务

合并编译技术介绍 - 版本管理

最终一致性 ⇨ 镜像平台配置好版本，下次上游上线带上去 ／ 流水线通知到群

强一致性 ⇨ 一条流水线同时升级上下游两个服务 ／ 拥有上游权限的团队直接升级上游服务



镜像平台下游给上游修改默认版本

上游选择依赖的下游版本

合并编译技术介绍 - 版本管理

下游查看上游被合并编译集群及其合并的版本

案例分析

- 期望获得性能收益的服务
- 重 SLA 类型的服务
- 其他场景

如何找到适合合并的服务

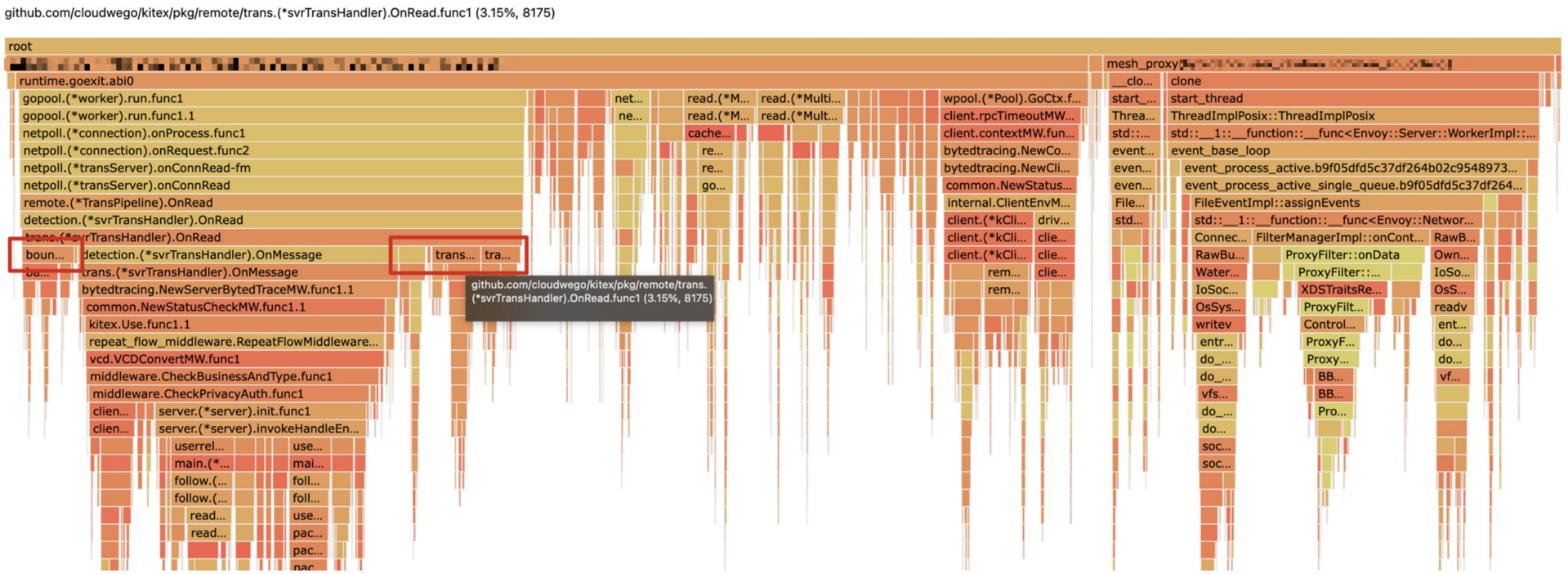
期望获得性能收益

- 流量调用关系：从 Trace 开销分析
- 服务大小
- 火焰图编解码开销
- 服务类型：缓存、固定开销类型的服务不适合合并

上游	下游	Client QPS	Client Ratio	Server QPS	Server Ratio	上游 Quota	下游 Quota	服务类型
api.comment	comment.biz	14299551	0,197	2949886	0,962	4.4w	5.5w	计算型

如何找到适合合并的服务

火焰图编解码开销



收益估算：下游 CPU Quota * (编解码开销 + 服务治理中间件开销) * 2　　收益：Quota 4.4w core，接口时延 avg 10ms

如何找到适合合并的服务

重 SLA 类型的服务

- 业务场景分析
- 大上游 + 小下游

上游	下游	Client QPS	Client Ratio	Server QPS	Server Ratio	上游 Quota	下游 Quota	服务类型
price_engine	marketing_price	35863974	0,002	90122	0,985	44w	1.5k	计算型

防止因为流量突增导致下游雪崩

期望改造成 SDK 的服务

- 中台化的服务

总结与展望

如何找到适合合并的服务

合并编译 — 探索研发效率和运行效率的最大化

- 降低部署期微服务数量，提升调用效率，降低成本
- 最大限度支持任意数量服务可逆聚合
- 最大程度降低非业务损耗

单体和微服务架构并不是绝对的，探索下一代服务架构

InfoQ 极客传媒

已结束 北京站 2000 人

QCon 全球软件开发大会 暨智能软件开发生态展

架构演进 | AIGC
研发效能 | 向量数据库

6月14-15日 深圳站 1000 人

ArchSummit 全球架构师峰会 暨智能软件开发生态展

业务架构 | 高可用
AI 架构 | 架构师成长

10月18-19日 上海站 1500 人

QCon 全球软件开发大会 暨智能软件开发生态展

AIGC | AI 原生应用
软件工程 | 金融科技

4月 5月 6月 8月 10月 12月

5月17-18日 北京站 2000 人

AiCon 全球人工智能开发与应用大会 暨大模型应用生态展

通用大模型 | AI Agent
LLMOps | AI Infra

8月16-17日 上海站 1000 人

FCon 全球金融科技大会 暨智能软件开发生态展

金融大模型 | 核心系统
数字化转型 | 数据安全

12月20-21日 北京站 1000 人

ArchSummit 全球架构师峰会 暨智能软件开发生态展

未来架构 | 高可用高并发
AI 应用 | 架构师思维

极客邦科技 2024 年会议规划

促进软件开发及相关领域知识与创新的传播

访问大会官网

参会咨询

THANKS

大模型正在重新定义软件

Large Language Model Is Redefining The Software