「半空」富脚手架模式

字节 Go2Rust 工程落地实践

演讲人：范广宇

2025.9.20

目录 | Contents

Part 01
背景
Go2Rust 的收益与挑战

Part 02
富脚手架模式
Go2Rust 迁移的完整解决方案

Part 03
未来展望
未来工作展望

01 背景

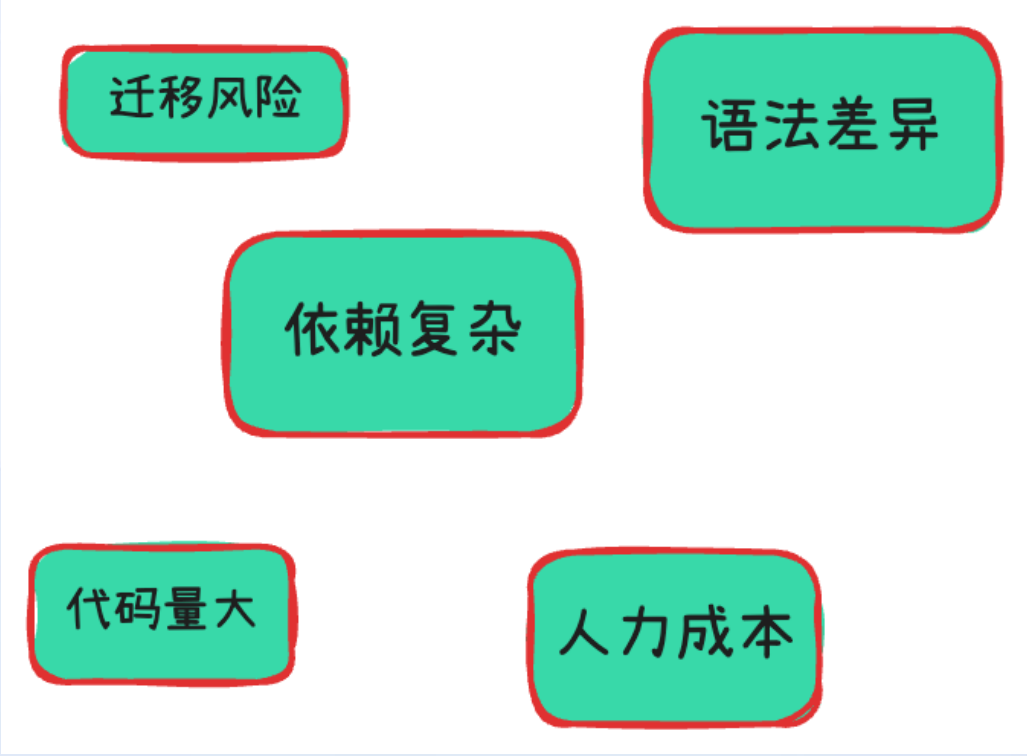
Go2Rust 的收益与挑战，LLM 和 ABCoder 为 Go2Rust 带来全新解决方案

Rust 迁移的背景

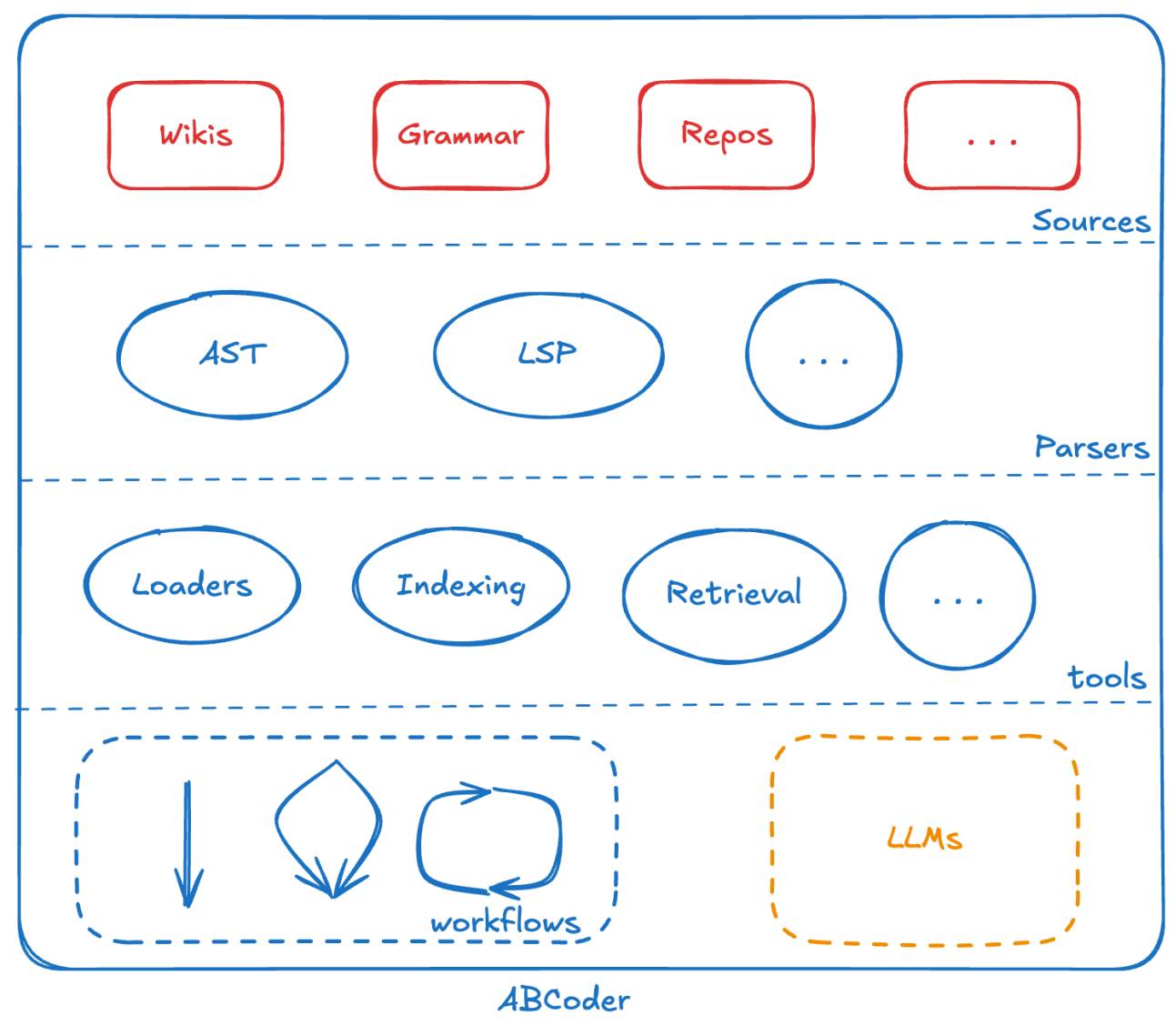
Golang -> Rust 迁移收益

- Golang 服务在迁移至 Rust 后，取得 **40~50%** 的 **CPU 收益，10~70%** 的 **P99 时延**
- Golang 服务在迁移至 Rust 后，内存安全性、服务稳定性、运维成本均得到收益

Golang -> Rust 迁移挑战

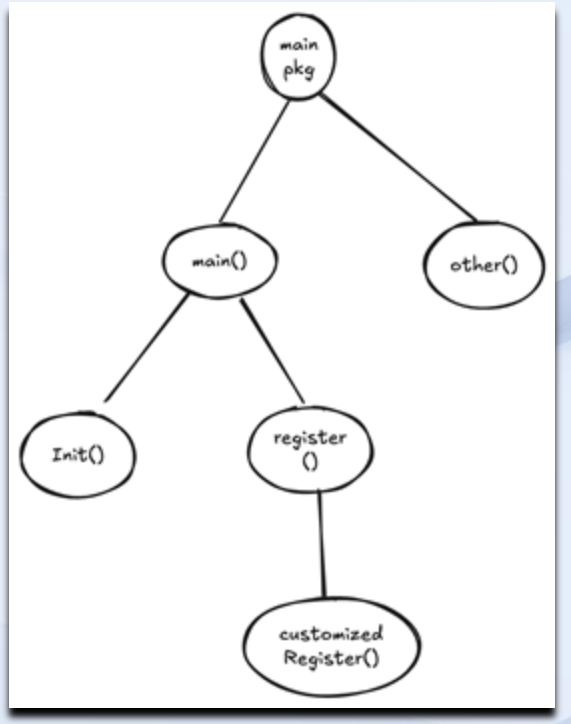


ABCoder: AI-Based Coder



ABCoder：一套 LLM 驱动的项目理解并应用的编程语言解决方案，它拥有一套完整的工作流，包括代码解析器、知识库构建、常用AI 工具等

核心能力：将代码仓库抽象为一棵具有节点调用关系的语法树。



Go2Rust 迁移的发展与变迁

「半空」：ABCoder X LLM 的 Go2Rust 解决方案

富脚手架模式：Go2Rust 一站式解决方案

- 目标：**为用户的 Go 项目交付一套 100% 可编译的、具有原始业务逻辑的 Rust 脚手架（0-1）**，用户可“开箱即用”，配合半空插件快速迁移
- 特点：
  - 依赖翻译 & 半径控制：**依赖级联翻译**，避免阻塞迁移；控制依赖深度，收敛依赖层数
  - 编译优先：优先保证整体脚手架**可编译**
  - IDE 亲和：提供 IDE 插件，配合用户完成迁移准备、业务迁移、持续迭代的工作

02

富脚手架模式

全面迈向 Go2Rust 迁移"自动档"时代的完整解决方案

人工 Golang 迁移 Rust 的流程

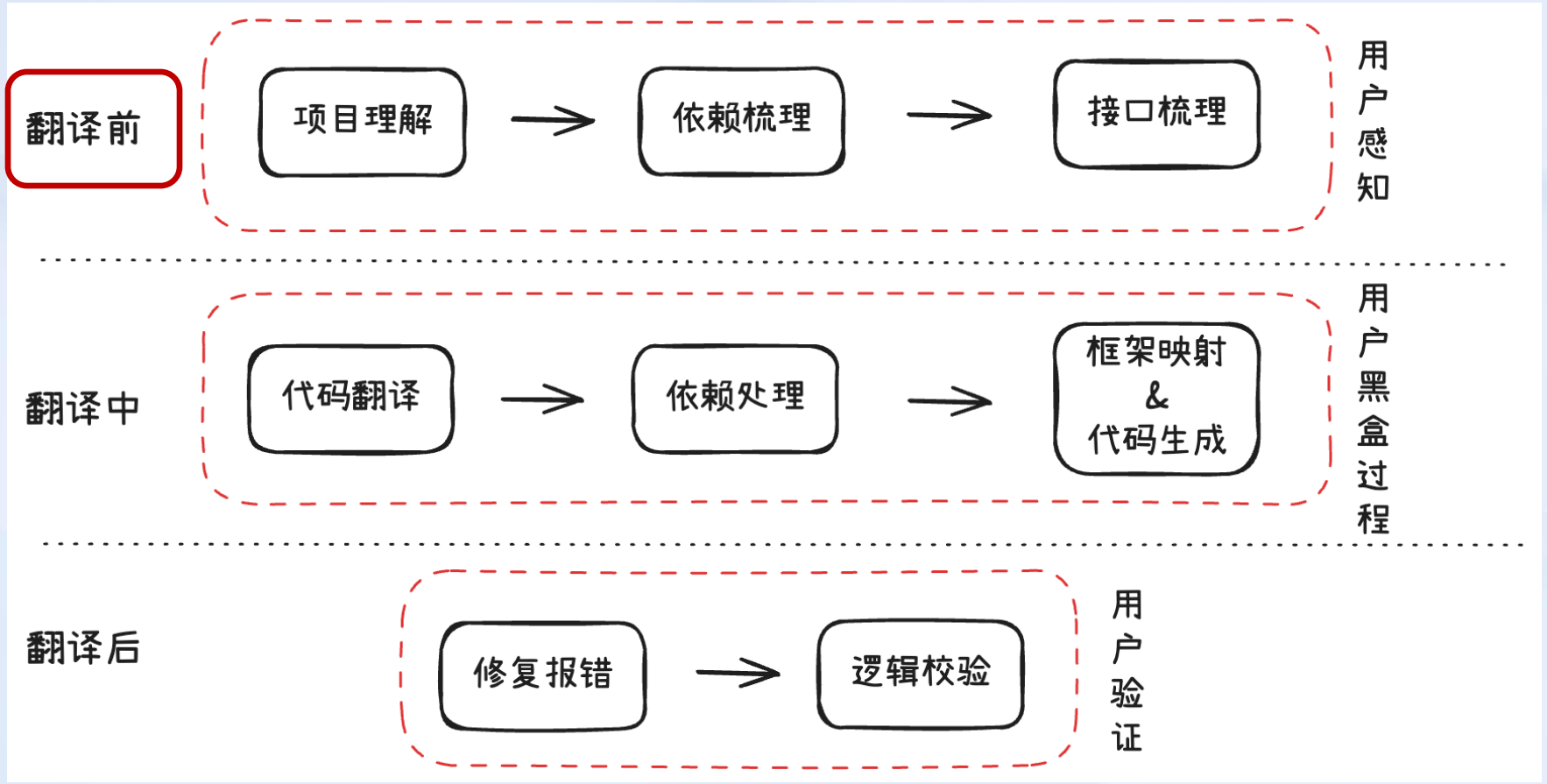
Go2Rust 迁移流程	说明	半空解决方案
项目立项	• 明确迁移项目的可行性、资源分配和目标等	
项目梳理	• 全面梳理项目的业务逻辑、接入的组件以及代码的边缘情况，为制定迁移方案做好充分准备	• 自动梳理、制定方案
技术调研 & 选项	• 调研 Rust 开发框架和组件生态，确保能够快速重写现有功能并持续维护	• 自动映射组件
人力规划	• 评估团队对 Rust 的熟悉程度，确保团队能够持续投入 Rust 的开发	• 共同完成
项目开发	• 实际投入人力进行项目整体迁移	• 自动迁移
项目验收	• 测试、上线	
持续迭代	• 后续使用 Rust 对项目进行持续开发	• 提供 IDE 插件

富脚手架模式的展示形式

- 以 IDE 插件形式对外提供服务
  - 产品融入现代智能 IDE(Trae、VSCode)
  - 所有交互通过 IDE 完成

- 一站式的Go2Rust解决方案
  - 翻译前：项目梳理、依赖分析
  - 翻译中：自动迁移 Golang 到 Rust
  - 翻译后：自动修错，高质量交付

富脚手架模式的执行流程



翻译前：项目迁移的“全自动规划书”

1. 项目深度理解文档

一、github.com/cloudwego/biz-demo/easy_note ...

二、接口介绍

- 1. NoteServiceImpl.QueryNote
- 2. UserServiceImpl.CreateUser
- 3. NoteServiceImpl.DeleteNote
- 4. NoteServiceImpl.UpdateNote
- 5. NoteServiceImpl.CreateNote
- 6. NoteServiceImpl.MGetNote
- 7. UserServiceImpl.CheckUser
- 8. UserServiceImpl.MGetUser

三、Package 详情

- 1. cloudwego/biz-demo/easy_note/api_request
- 2. cloudwego/biz-demo/easy_note/api_request/api_...
- 3. cloudwego/biz-demo/easy_note/cmd/api
- 4. cloudwego/biz-demo/easy_note/cmd/api/hertz_h...
- 5. cloudwego/biz-demo/easy_note/cmd/api/hertz_h...
- 6. cloudwego/biz-demo/easy_note/cmd/api/hertz_ro...
- 7. cloudwego/biz-demo/easy_note/cmd/api/hertz_ro...

一、github.com/cloudwego/biz-demo/easy_note 项目概述

项目名称：`abcoder/biz_demo`

简述：

这是一个基于Hertz和Kitex框架的简易笔记系统微服务架构，提供完整的用户认证和笔记管理功能。系统采用分层设计，包含API网关层、业务服务层和数据访问层，支持JWT身份验证、分布式追踪和ETCD服务发现。API网关通过Hertz框架提供RESTful接口，业务服务(用户/笔记)通过Kitex实现RPC调用，数据层使用GORM操作MySQL。系统具备标准化错误处理、日志记录和性能监控能力，适用于个人笔记管理、协作编辑等场景。

序号	包名	描述
1	cloudwego/biz-demo/easy_note/api_request	简易笔记系统API模块，处理用户认证和笔记CRUD操作，支持命令行交互。
2	cloudwego/biz-demo/easy_note/api_request/api_service	HTTP客户端库，提供笔记服务API交互功能，支持用户和笔记管理及HTTP请求处理。
3	cloudwego/biz-demo/easy_note/cmd/api	基于Hertz框架的API服务，集成RPC、JWT和日志，提供路由注册和HTTP服务。
4	cloudwego/biz-demo/easy_note/cmd/api/hertz_handler	HTTP API处理器，提供ping/pong测试接口和JSON响应功能。
5	cloudwego/biz-demo/easy_note/cmd/api/hertz_handler/demoapi	基于Hertz框架的笔记服务API，提供用户认证、笔记管理和统一响应处理功能。
6	cloudwego/biz-demo/easy_note/cmd/api/hertz_router	Hertz框架路由注册模块，自动将IDL生成的路由注册到服务器。

接口函数实现

NoteServiceImpl.QueryNote 是一个查询笔记的服务方法。该方法首先初始化响应对象，然后验证请求参数的有效性，如果无效则返回参数错误响应。接着检查分页限制，如果未设置则使用默认值。然后调用 QueryNoteService 查询笔记列表和总数，如果查询过程中出现错误则返回错误响应。最后构造成功响应，填充笔记列表和总数，并返回响应对象。

输入参数：

- **s**: NoteServiceImpl 的实例，实现了笔记服务的接口
- **ctx**: context.Context 类型的参数，用于传递上下文信息
- **req**: 查询笔记的请求参数，包含用户ID、搜索关键字、分页限制和偏移量

输出参数：

- **resp**: 查询笔记的响应，包含笔记列表、总数和基础响应信息
- **err**: 错误信息，类型为 error

数据流：

- **req**:
  - 验证请求参数的有效性
  - 将请求参数传递给 QueryNoteService 用于查询笔记列表和总数

副作用：

- 执行数据库查询，可能对数据库产生负载

翻译前：项目迁移的“全自动规划书”

2. 依赖梳理

1. 直接依赖

> 此列表是目标仓库的第一层依赖，会在代码中直接调用

依赖模块	依赖节点	Rust是否已经实现
go.opentelemetry.io/otel/trace@v1.27.0	- Node Name: go.opentelemetry.io/otel/trace#SpanFromContext -- FUNC - Node Name: go.opentelemetry.io/otel/trace#TraceID.String -- FUNC - Node Name: go.opentelemetry.io/otel/trace#Span.SpanContext -- FUNC	
gorm.io/driver/mysql@v1.4.4	- Node Name: gorm.io/driver/mysql#Open -- FUNC	
gorm.io/gorm@v1.25.10	- Node Name: gorm.io/gorm#DB.Delete -- FUNC - Node Name: gorm.io/gorm#DB.Where -- FUNC	
gorm.io/plugin/opentelemetry@v0.1.4	- Node Name: gorm.io/plugin/opentelemetry/logging/logrus#NewWriter -- FUNC - Node Name: gorm.io/plugin/opentelemetry/tracing#NewPlugin -- FUNC	
~~github.com/cloudwego/hertz@v0.9.5~~	- Node Name: github.com/cloudwego/hertz/pkg/app#RequestContext.BindAndValidate -- FUNC - Node Name: github.com/cloudwego/hertz/pkg/app#RequestContext.JSON -- FUNC	
~~github.com/cloudwego/kitex@v0.11.3~~	- Node Name: github.com/cloudwego/kitex/server#WithServiceAddr -- FUNC - Node Name: github.com/cloudwego/kitex/server#WithRegistry -- FUNC - Node Name: github.com/cloudwego/kitex/server#WithLimit -- FUNC - Node Name: github.com/cloudwego/kitex/server#WithMuxTransport -- FUNC	
~~github.com/golang-jwt/jwt/v4@v4.5.2~~	- Node Name: github.com/golang-jwt/jwt/v4#WithJSONNumber -- FUNC - Node Name: github.com/golang-jwt/jwt/v4#ParserOption -- TYPE	

2. 间接依赖

> 内部依赖列表是目标仓库的第二层依赖，是直接依赖的仓库代码中直接调用的当前库的节点
>
> 外部依赖列表是目标仓库的第三层依赖，是直接依赖的仓库代码中间接调用的第三方库的节点

模块	内部依赖	外部依赖
~~github.com/cloudwego/hertz@v0.9.5~~	Node Name: **github.com/cloudwego/hertz/pkg/app#RequestContext.BindAndValidate** -- FUNC - dep 1: github.com/cloudwego/hertz/pkg/app/server/binding#Binder.BindAndValidate - dep 2: github.com/cloudwego/hertz/pkg/network#Reader - dep 3: github.com/cloudwego/hertz/pkg/network#Writer	- **[]github.com/cloudwego/hertz/pkg/comm** **Option** -- UNKNOWN - **error** -- UNKNOWN - **func(options* *github.com/cloudwego/hertz/pkg/common/c* *github.com/cloudwego/hertz/pkg/network.Tr* Interface
~~github.com/cloudwego/kitex@v0.11.3~~	Node Name: **github.com/cloudwego/kitex/client#WithClientBasicInfo** -- FUNC - dep 1: github.com/cloudwego/kitex/pkg/rpcinfo#EndpointBasicInfo - dep 2: github.com/cloudwego/kitex/pkg/utils#Slice - dep 3: github.com/cloudwego/kitex/internal/client#Option - dep 4: github.com/cloudwego/kitex/client#Option - dep 5: github.com/cloudwego/kitex/pkg/diagnosis#ProbeFunc - dep 6: github.com/cloudwego/kitex/pkg/diagnosis#Service - dep 7: github.com/cloudwego/kitex/pkg/loadbalance/lbcache#Options - dep 8: github.com/cloudwego/kitex/pkg/rpcinfo#Timeouts - dep 9: github.com/cloudwego/kitex/pkg/rpcinfo#StreamConfig - dep 10: github.com/cloudwego/kitex/pkg/rpcinfo#RPCConfig - dep 11: github.com/cloudwego/kitex/internal/client#ConfigLocks - dep 12: github.com/cloudwego/kitex/internal/configutil#OptionOnce - dep 13: github.com/cloudwego/kitex/pkg/stats#Event - dep 14: github.com/cloudwego/kitex/internal/client#StreamOptions	- **github.com/bytedance/gopkg/cloud/circ** -- TYPE - **github.com/bytedance/gopkg/collection/s** **p** -- TYPE - **github.com/cloudwego/localsession/backu** **r** -- TYPE - **github.com/cloudwego/localsession/bac** TYPE - **golang.org/x/sync/singleflight#Group** --

翻译前：项目迁移的“全自动规划书”

3. 接口梳理

1 NoteServiceImpl.CreateNote 接口

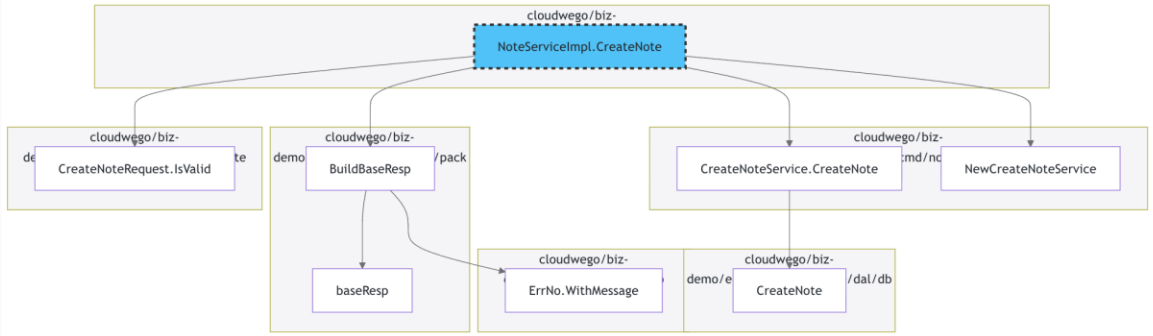
依赖当前仓库的节点数量：23
其中:

package	节点名	节点类型
github.com/cloudwego/biz-demo/easy_note/cmd/note/dal/db	CreateNote	FUNC
github.com/cloudwego/biz-demo/easy_note/cmd/note/dal/db	Note	TYPE
github.com/cloudwego/biz-demo/easy_note/kitex_gen/demouser	ErrCode	TYPE
github.com/cloudwego/biz-demo/easy_note/kitex_gen/demouser	ErrCode_SuccessCode	VAR
github.com/cloudwego/biz-demo/easy_note/kitex_gen/demouser	ErrCode_ServiceErrCode	VAR
github.com/cloudwego/biz-demo/easy_note/kitex_gen/demouser	ErrCode_ParamErrCode	VAR
github.com/cloudwego/biz-demo/easy_note/cmd/note/service	CreateNoteService	TYPE
github.com/cloudwego/biz-demo/easy_note/cmd/note/service	CreateNoteService.CreateNote	FUNC
github.com/cloudwego/biz-demo/easy_note/cmd/note/service	NewCreateNoteService	FUNC
github.com/cloudwego/biz-demo/easy_note/kitex_gen/demonote	CreateNoteRequest	TYPE
github.com/cloudwego/biz-demo/easy_note/kitex_gen/demonote	BaseResp	TYPE
github.com/cloudwego/biz-demo/easy_note/kitex_gen/demonote	CreateNoteResponse	TYPE
github.com/cloudwego/biz-demo/easy_note/kitex_gen/demonote	CreateNoteRequest.IsValid	FUNC

依赖三方仓库的节点数量：3
其中:

依赖模块	function 数量	type 数量	var 数量	总计
gorm.io/gorm@v1.25.10	2	1	0	3

函数调用链

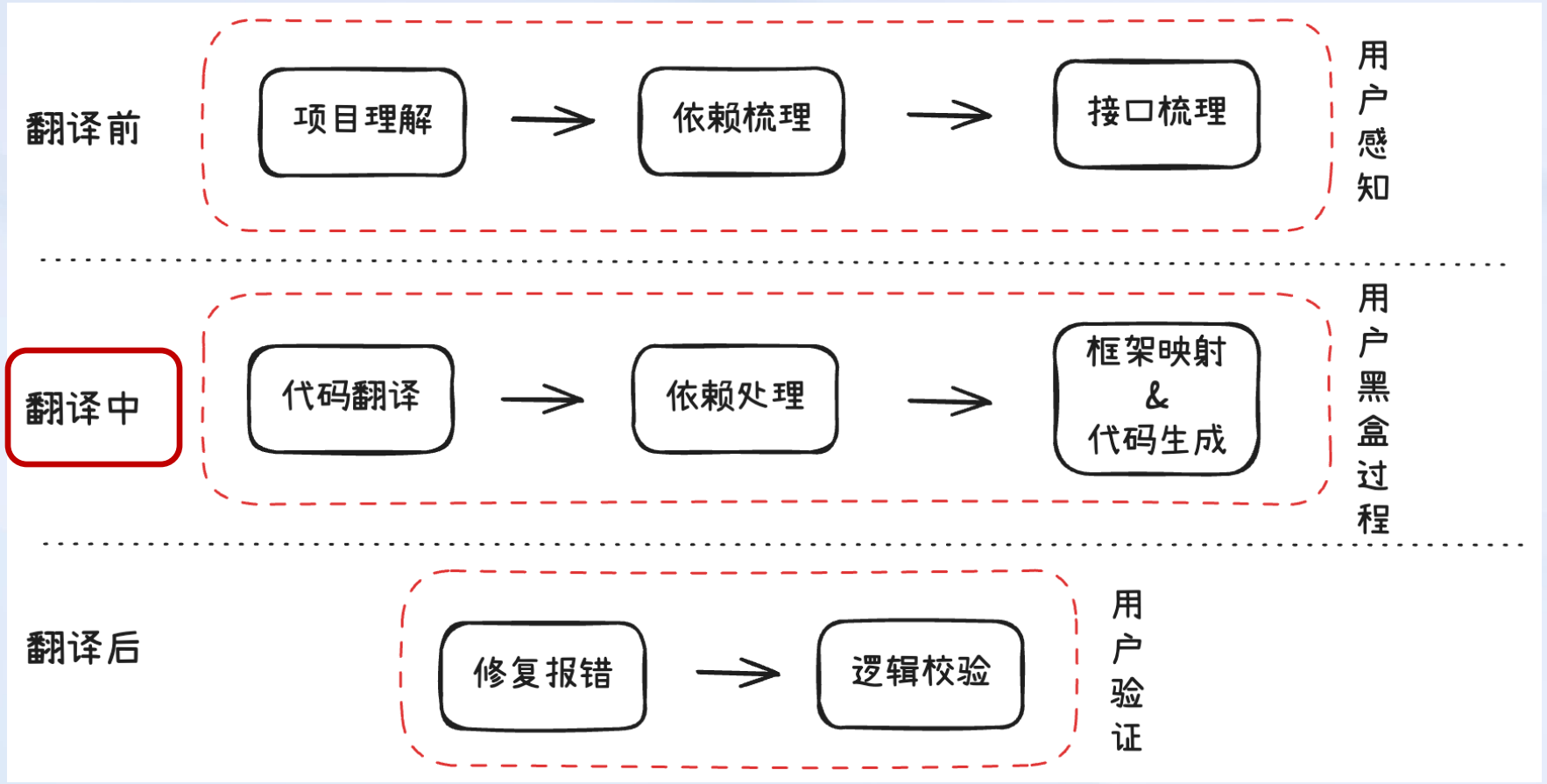


翻译前：项目迁移的“全自动规划书”

4. 迁移规划书

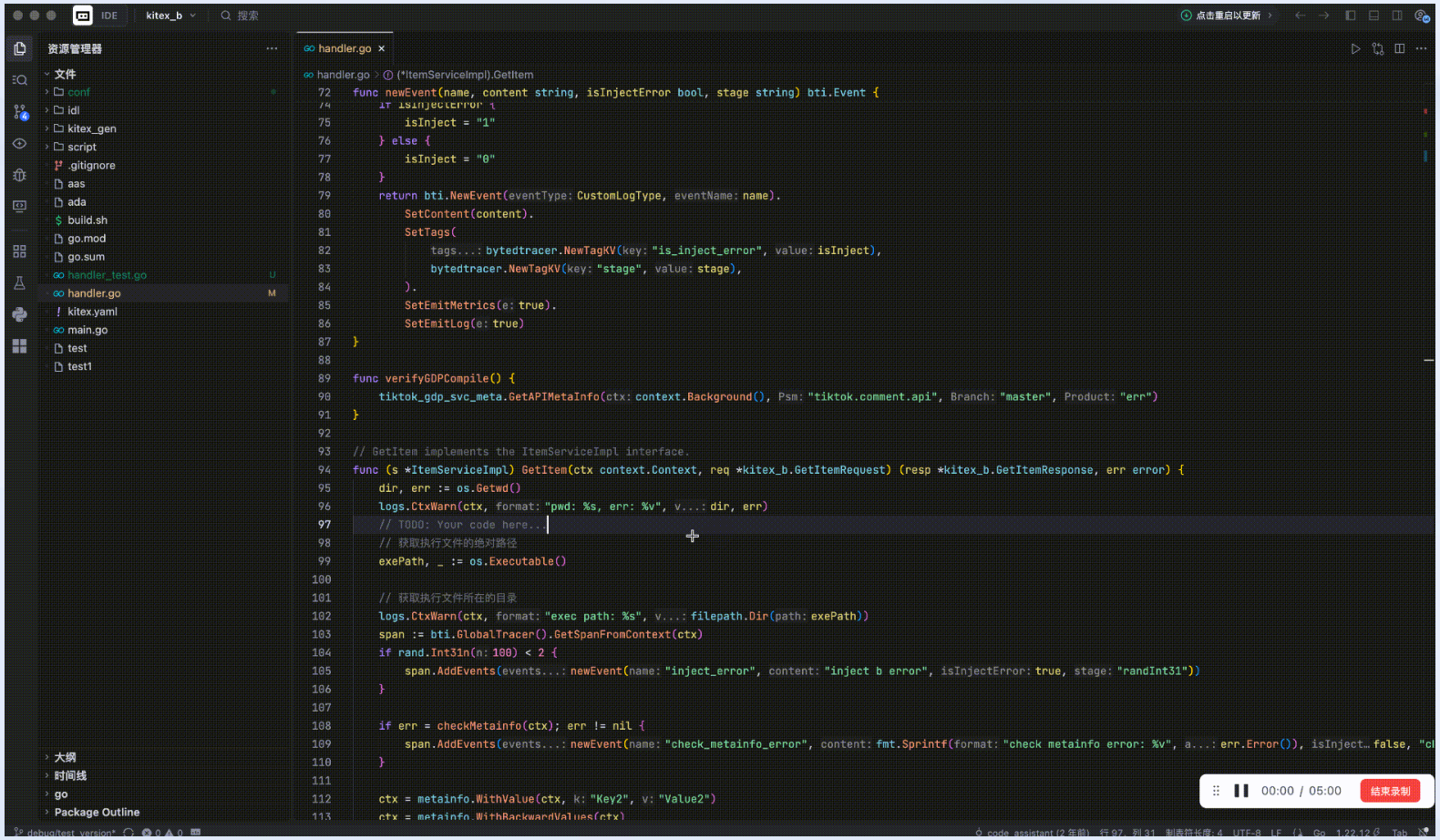
- 基于上述分析，自动生成规划书 & 分析翻译难度
- 与业务同学共同确定最终迁移的时间安排 & 人力规划

富脚手架模式的执行流程



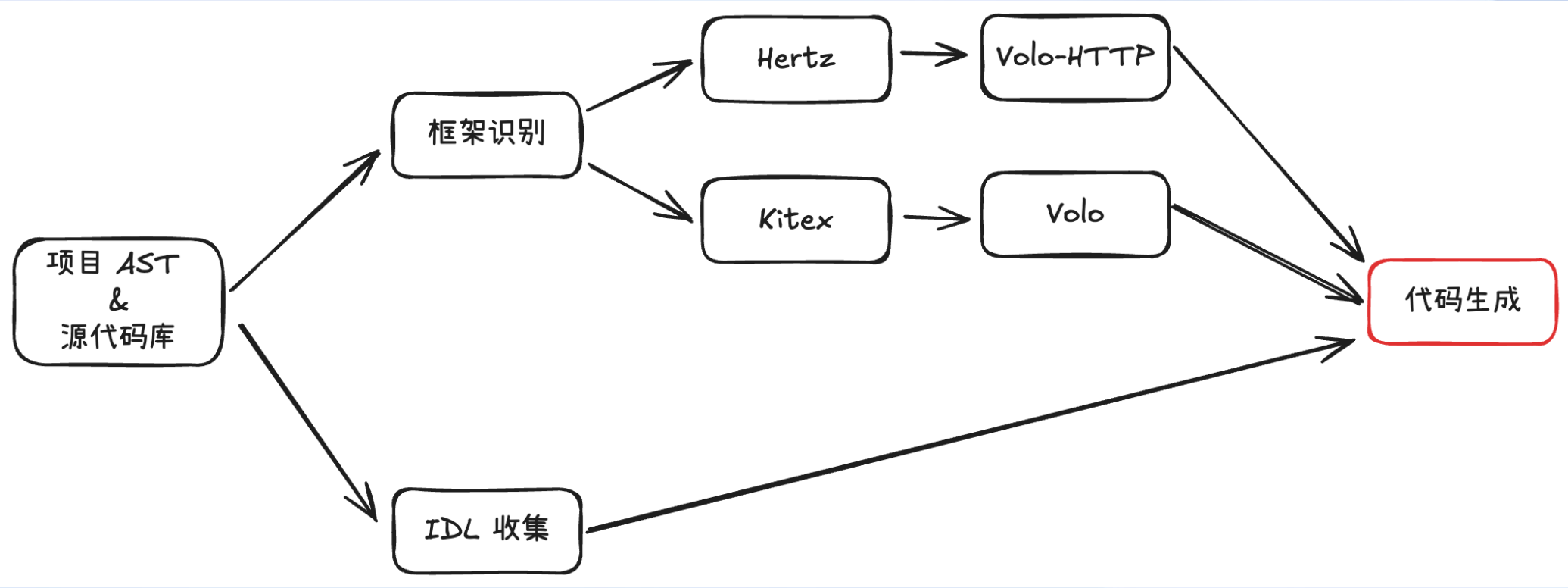
翻译中：全自动地 Go2Rust 流程

1. IDE 插件一键触发



翻译中：全自动地 Go2Rust 流程

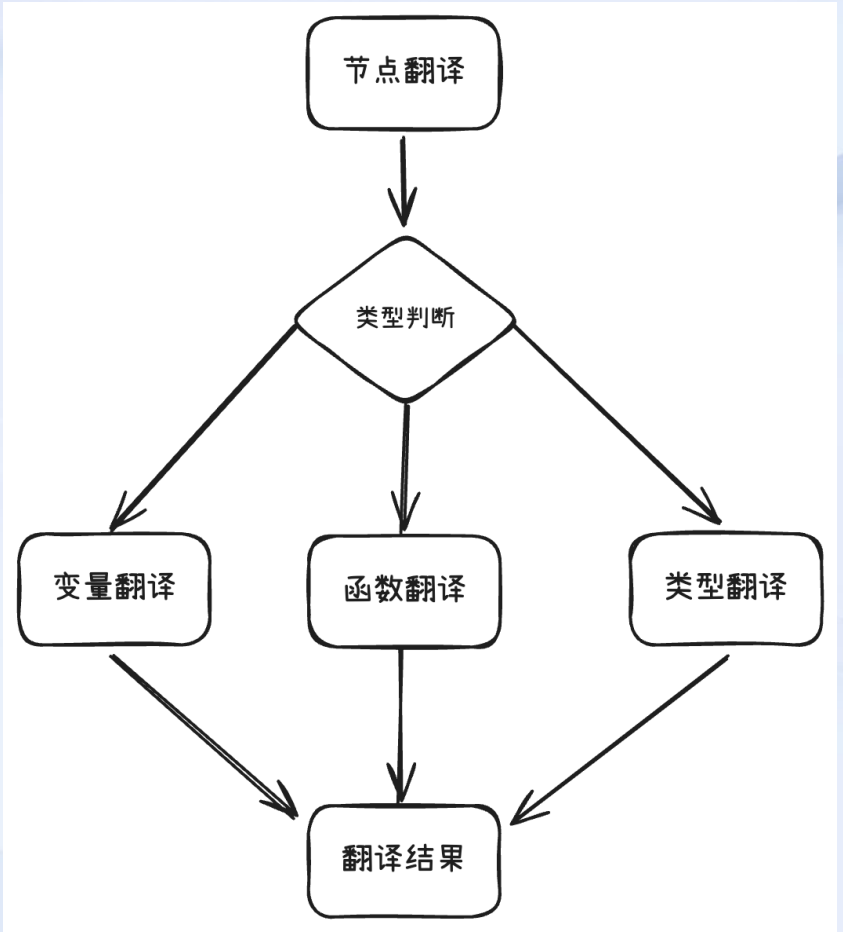
2. 框架映射 & 代码生成



翻译中：全自动地 Go2Rust 流程

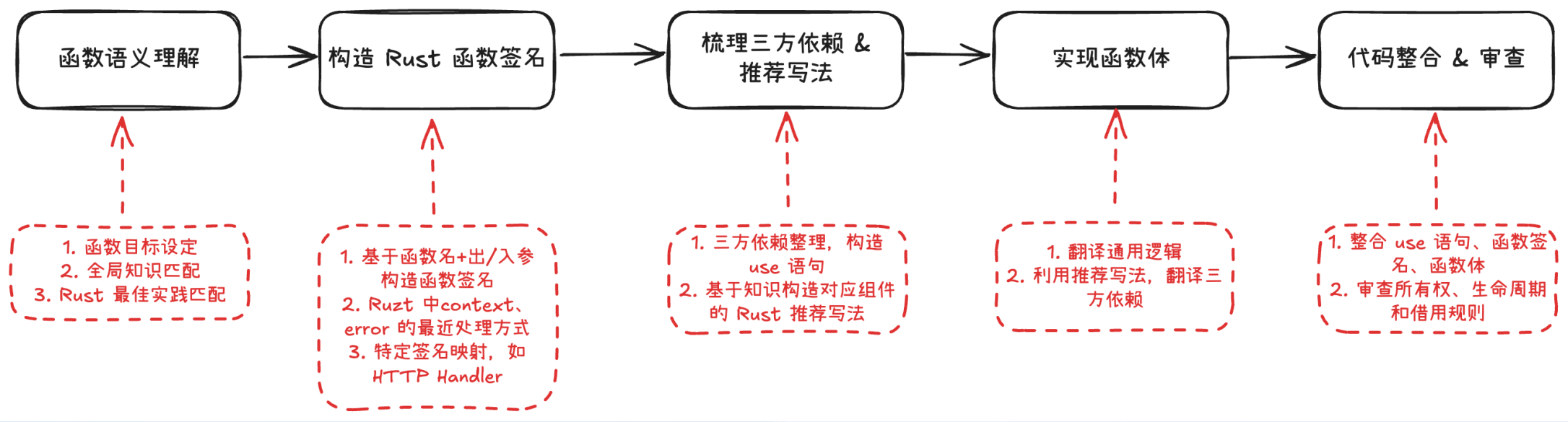
2. 节点翻译：

- 函数节点：业务逻辑抽象、LLM 意译翻译
- 变量节点：指定映射规则、LLM 意译翻译
- 类型节点：指定映射规则、LLM 意译翻译



翻译中：全自动地 Go2Rust 流程

2. 节点翻译： 函数翻译流程



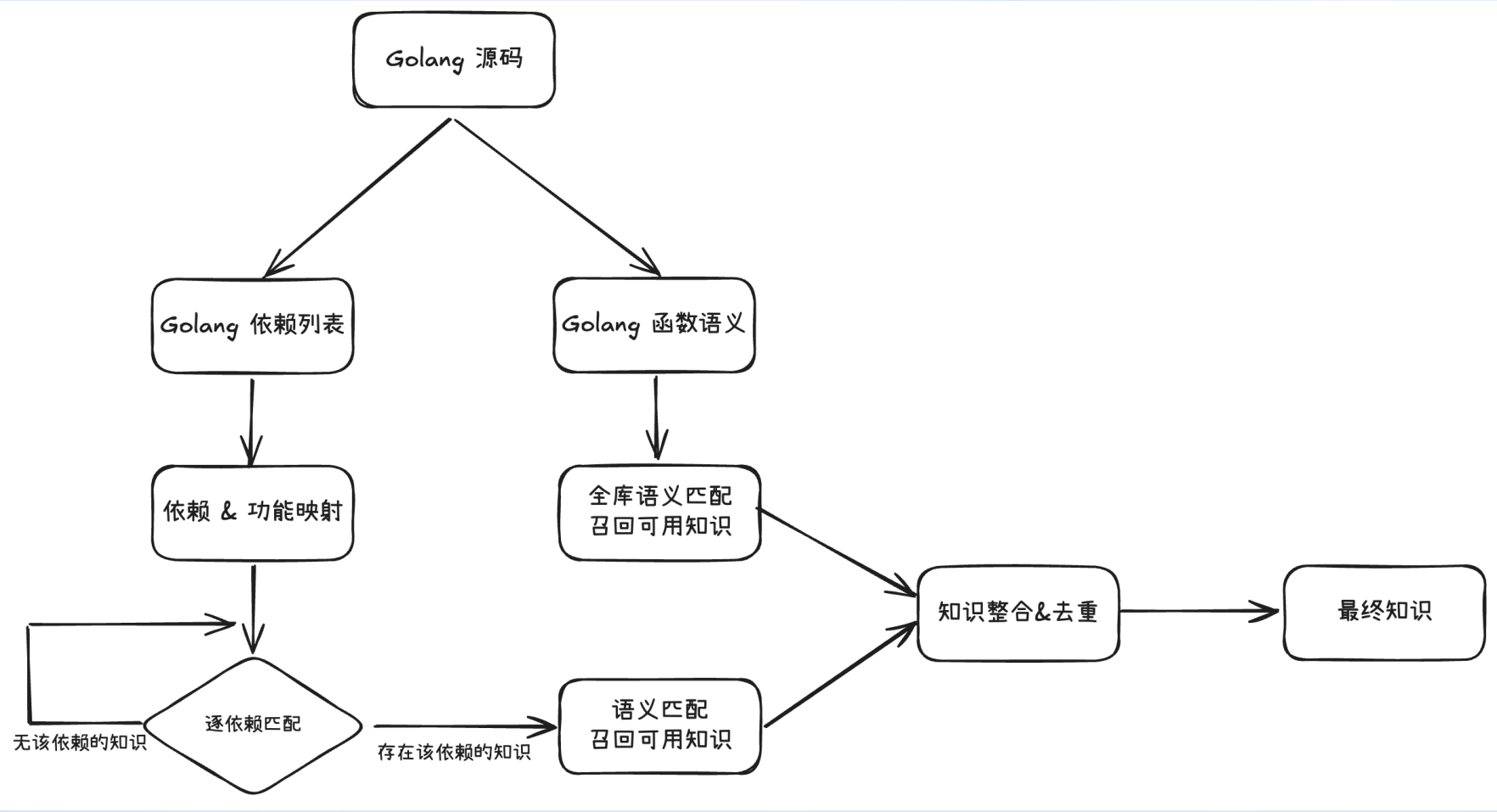
翻译中：全自动地 Go2Rust 流程

2. 节点翻译： 依赖处理方式

依赖类型	处理策略	技术手段
本地依赖	递归翻译	• 自底向上先翻译被依赖的节点，确保翻译函数前，其依赖的节点被翻译好
业务 SDK	级联翻译	• 提前识别并标记业务封装的 SDK • 同本地依赖的处理方式，递归翻译业务 SDK 节点 • 确保翻译函数前，其依赖的SDK节点被翻译好
内部基础组件	知识库映射	• 提前构建内部组件的 Rust 使用知识库 • 翻译函数前，如果使用了内部组件，则提前召回这些组件的 Rust 使用方法
开源组件	知识库映射 & 模型内置知识	• 如果知识库已有知识，则召回这些使用方法 • 如果知识库没有知识，则使用模型内置知识，利用模型本身的能力进行映射

翻译中：全自动地 Go2Rust 流程

2. 节点翻译： 知识召回



翻译中：全自动地 Go2Rust 流程

2. 节点翻译： 示例

```go
import (
    "context"
    "errors"
    "fmt"
    "strconv"

    "code.byted.org/xxx/logs" // 内部 log 组件库
    "code.byted.org/xxx/yyy/constant" // 本地依赖
    "code.byted.org/xxx/yyy/kitex_gen/example"  // 本地依赖
    "code.byted.org/xxx/yyy/redisdao" // 本地依赖
)

func GetIDFromRedis(ctx context.Context, liveID example.LiveIdEnum, accountId int64) (int64, error) {
    redisClient := redisdao.GetRedisClient(ctx, liveID)
    if redisClient == nil {
        return 0, errors.New("redis client is nil")
    }
    // RedisKey := "xxx_yyy_success:x-%d-y-%d"
    lockKey := fmt.Sprintf(constant.RedisKey, liveID, accountId)
    val, err := redisClient.Get(lockKey).Result()
    if err ≠ nil {
        logs.CtxInfo(ctx, "redis get id from redis failed, error = %v , accountId = %v", err,
accountId)return 0, err
    }
    ID, err := strconv.ParseInt(val, 10, 64)
    if err ≠ nil {
        logs.CtxError(ctx, "Parse ID error: %v", err)
        return 0, err
    }
    return ID, nil
}
```

提取翻译信息

1. Golang 源代码

- 函数本身

2. 依赖列表

- xxx/logs：内部 log 库
- xxx/yyy/constant： 本地依赖
- xxx/yyy/kitex_gen/example：本地依赖
- xxx/yyy/redisdoc：本地依赖
- xxx/yyy/redis: 内部 Redis 组件

3. 知识召回

- logs：在 rust 中使用 “tracing”打印日志
- redis：在 rust 中使用 “byted_redis”操作 Redis

翻译中：全自动地 Go2Rust 流程

2. 节点翻译： 示例

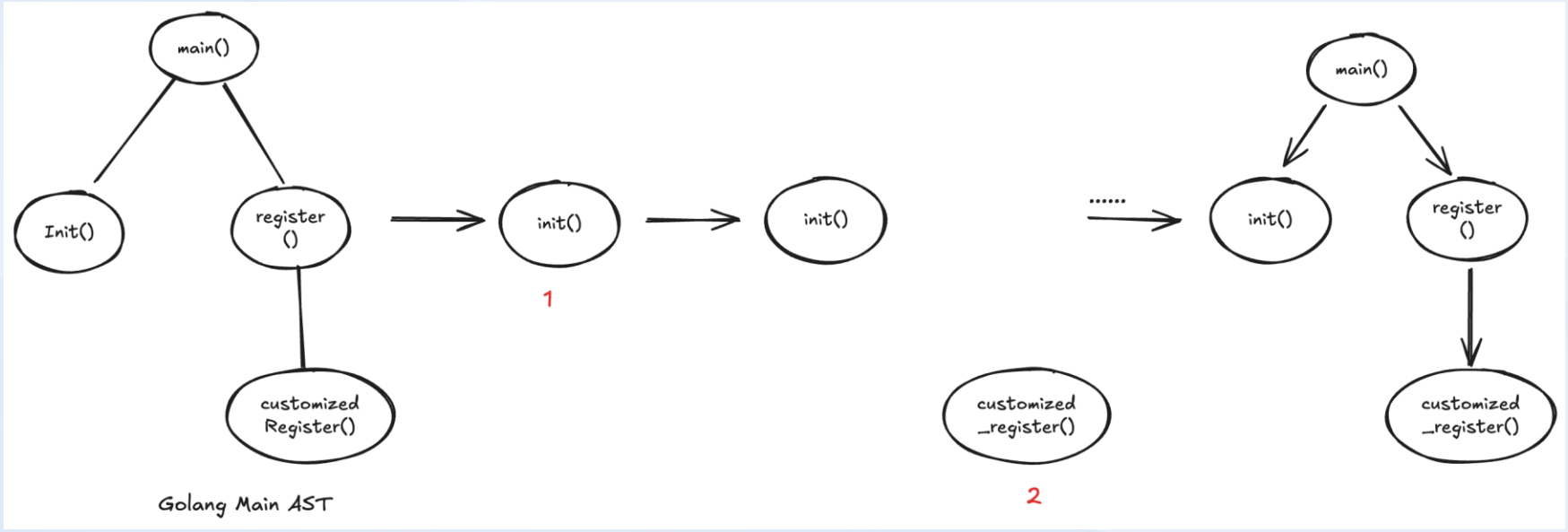
```go
import (
    "context"
    "errors"
    "fmt"
    "strconv"

    "code.byted.org/xxx/logs" // 内部 log 组件库
    "code.byted.org/xxx/yyy/constant" // 本地依赖
    "code.byted.org/xxx/yyy/kitex_gen/example"  // 本地依赖
    "code.byted.org/xxx/yyy/redisdao" // 本地依赖
)

func GetIDFromRedis(ctx context.Context, liveID example.LiveIdEnum, accountId int64) (int64, error) {
    redisClient := redisdao.GetRedisClient(ctx, liveID)
    if redisClient == nil {
        return 0, errors.New("redis client is nil")
    }
    // RedisKey := "xxx_yyy_success:x-%d-y-%d"
    lockKey := fmt.Sprintf(constant.RedisKey, liveID, accountId)
    val, err := redisClient.Get(lockKey).Result()
    if err != nil {
        logs.CtxInfo(ctx, "redis get id from redis failed, error = %v , accountId = %v", err,
accountId)return 0, err
    }
    ID, err := strconv.ParseInt(val, 10, 64)
    if err != nil {
        logs.CtxError(ctx, "Parse ID error: %v", err)
        return 0, err
    }
    return ID, nil
}
```

```rust
use anyhow::{Context, Result};
use lust_gen::ies_live;
use crate::redisdao::get_redis_client;
use crate::constant::REDIS_KEY;
use tracing::{info, error};

pub async fn get_recent_packet_id(live_id: example::LiveIdEnum, account_id: i64) -> Result<i64> {
    // 1. 获取 Redis 客户端
    let redis_client = get_redis_client(live_id)
        .ok_or_else(|| anyhow::anyhow!("redis client is nil"))?;

    // 2. 构造 Redis 锁键
    // INFO: Rust 不允许 format 格式话使用常量，自动其复制过来
    let lock_key = format!("packet_id_recently_recharge_success:liveId-{}-accountId-{}", live_id,
account_id);
    // 3. 从 Redis 获取值
    let mut conn = redis_client.get_async_connection().await
        .context("获取 Redis 连接失败")?;

    let val: String = conn.get(&lock_key).await
        .map_err(|err| {
            // INFO: 根据 knowledge 规则，已将 'logs.CtxInfo' 替换为使用 'tracing::info'
            info!("redis get id failed, error = {:?}, accountId = {}", err, account_id);
            err
        })
        .context("从 Redis 获取 id 失败")?;

    // 4. 解析数据包ID
    let packet_id = val.parse::<i64>()
        .map_err(|err| {
            // INFO: 根据 knowledge 规则，已将 'logs.CtxError' 替换为使用 'tracing::error'
            error!("Parse packetId error: {:?}", err);
            err
        })
        .context("解析 packetId 失败")?;

    Ok(packet_id)
}
```

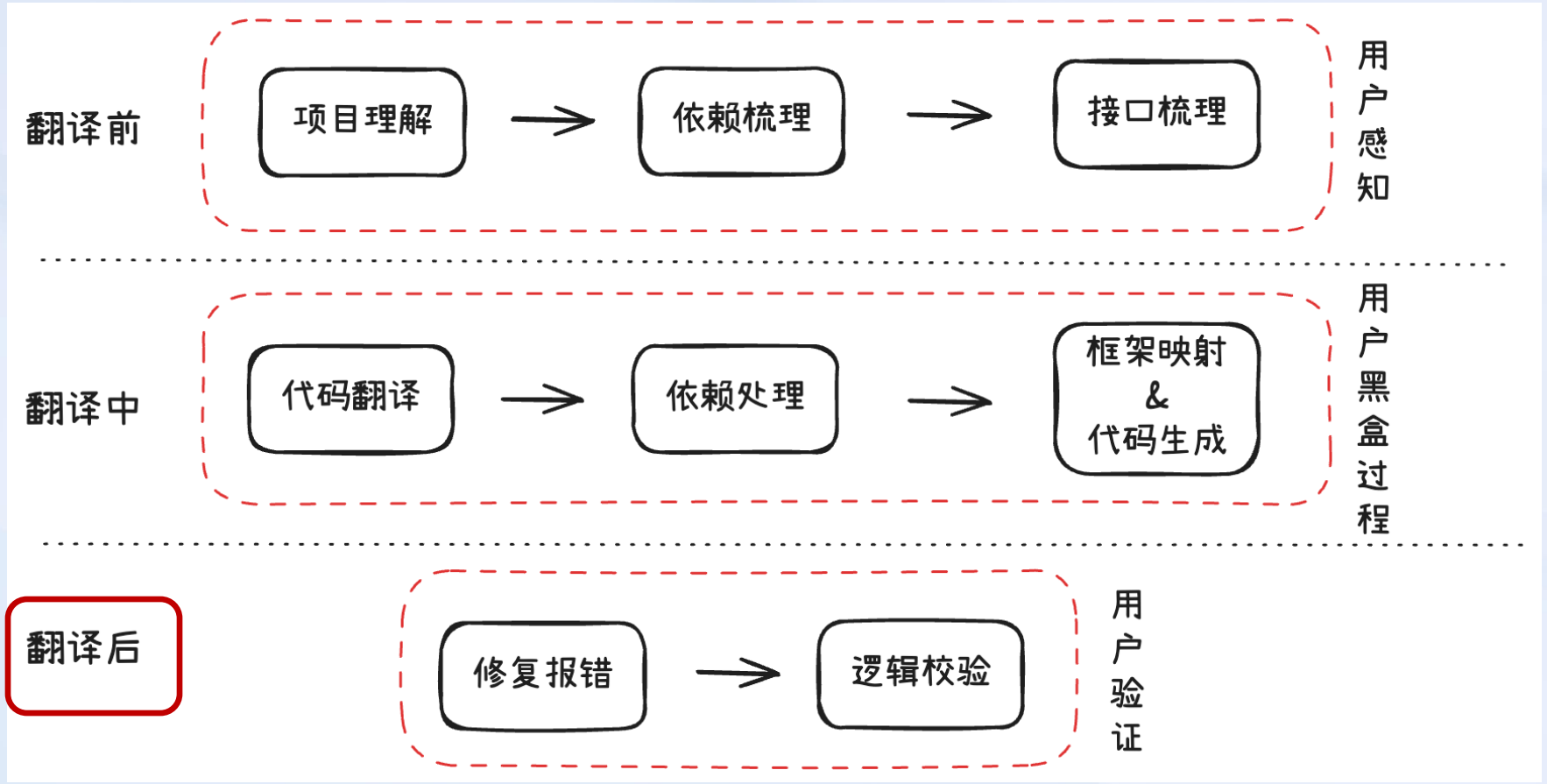
翻译中：全自动地 Go2Rust 流程

3. 接口链路翻译

- 基于梳理好地接口调用链路，自底向上地翻译每一个节点，直到调用链路全部翻译完毕



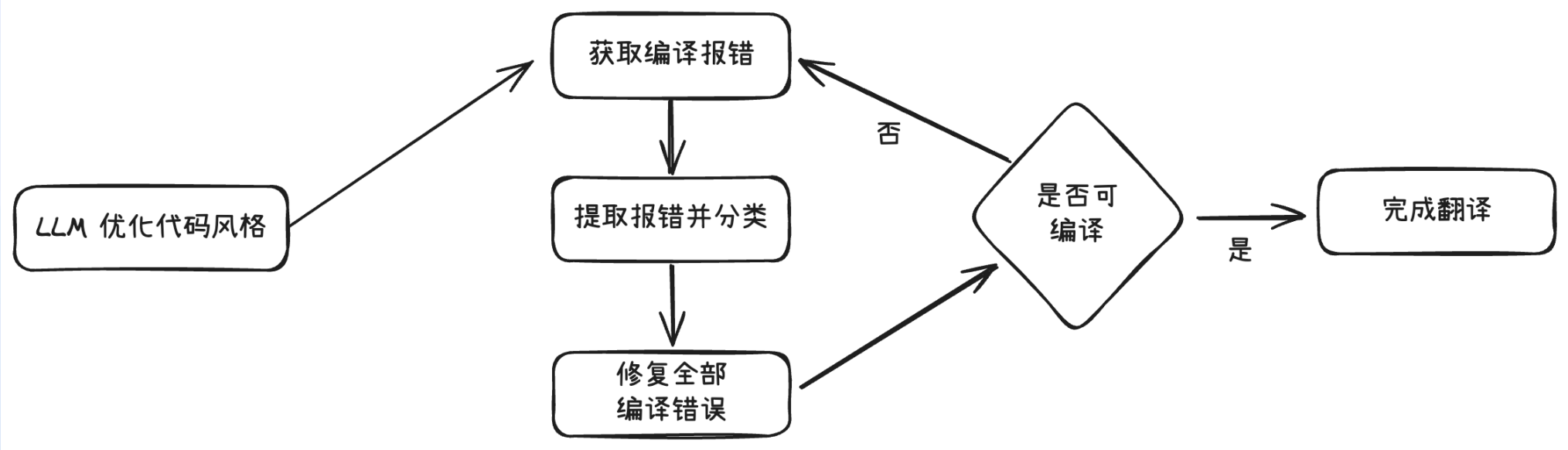
富脚手架模式的执行流程



翻译后：高质量的 Rust 代码交付

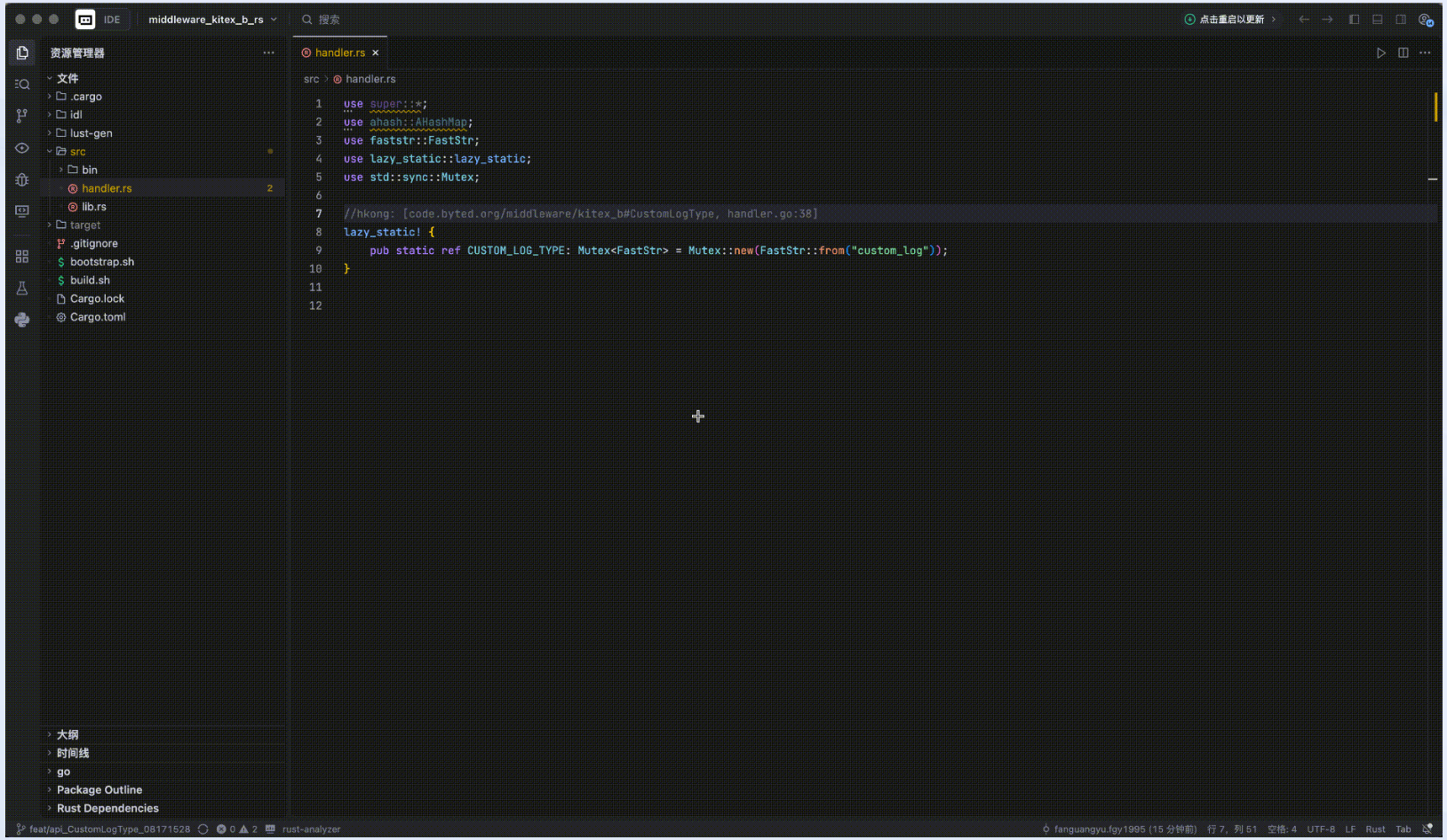
1. 基于 LLM Agent 驱动的代码质量优化

- 遍历函数，优化代码风格，使得翻译好的 Rust 代码符合最佳实践
- 获取 Rust 编译报错，使用 LLM 多轮修复，直至可编译状态



翻译后：高质量的 Rust 代码交付

2. 无缝逻辑校验：一键跳转回 Golang 源码



翻译后：高质量的 Rust 代码交付

3. 翻译质量回收

翻译项目名	package	branch_or_tag	总节点 ↓	compilation_score	优化后是否可编译
[redacted]	api_node[redacted]_MGetPackData	master	1413	0.833029	否
[redacted]	api_node[redacted]_GetInnerReachData	master	484	0.786700	是
[redacted]	api_node[redacted]MultiGetUserModel	master	413	0.667084	否
[redacted]	api_node_main	master	104	0.628663	是
[redacted]	api_node[redacted]QueryCDSGroupUpdatedFiles	master	52	0.827381	是
[redacted]	api_nod[redacted]_MGetPackData	master	33	0.778733	是
[redacted]	api_node[redacted]HasChannelDataStart	master	27	0.796537	是
[redacted]	api_node[redacted]buildShopAnchor	master	10	0.904762	是
[redacted]	api_node_main	master	5	1.000000	是

展望

01

Research 架构探索

提高翻译维度，发挥模型的能动性，实现 Go2Rust 重构级翻译

02

持续推广 & 优化

建立完善的用户行为反馈机制，基于用户的修改动作优化翻译质量

Q & A

THANKS