CloudWeGo

LLM 驱动 Go2Rust 代码迁移实践

范广宇

字节跳动研发工程师

2025/03/22

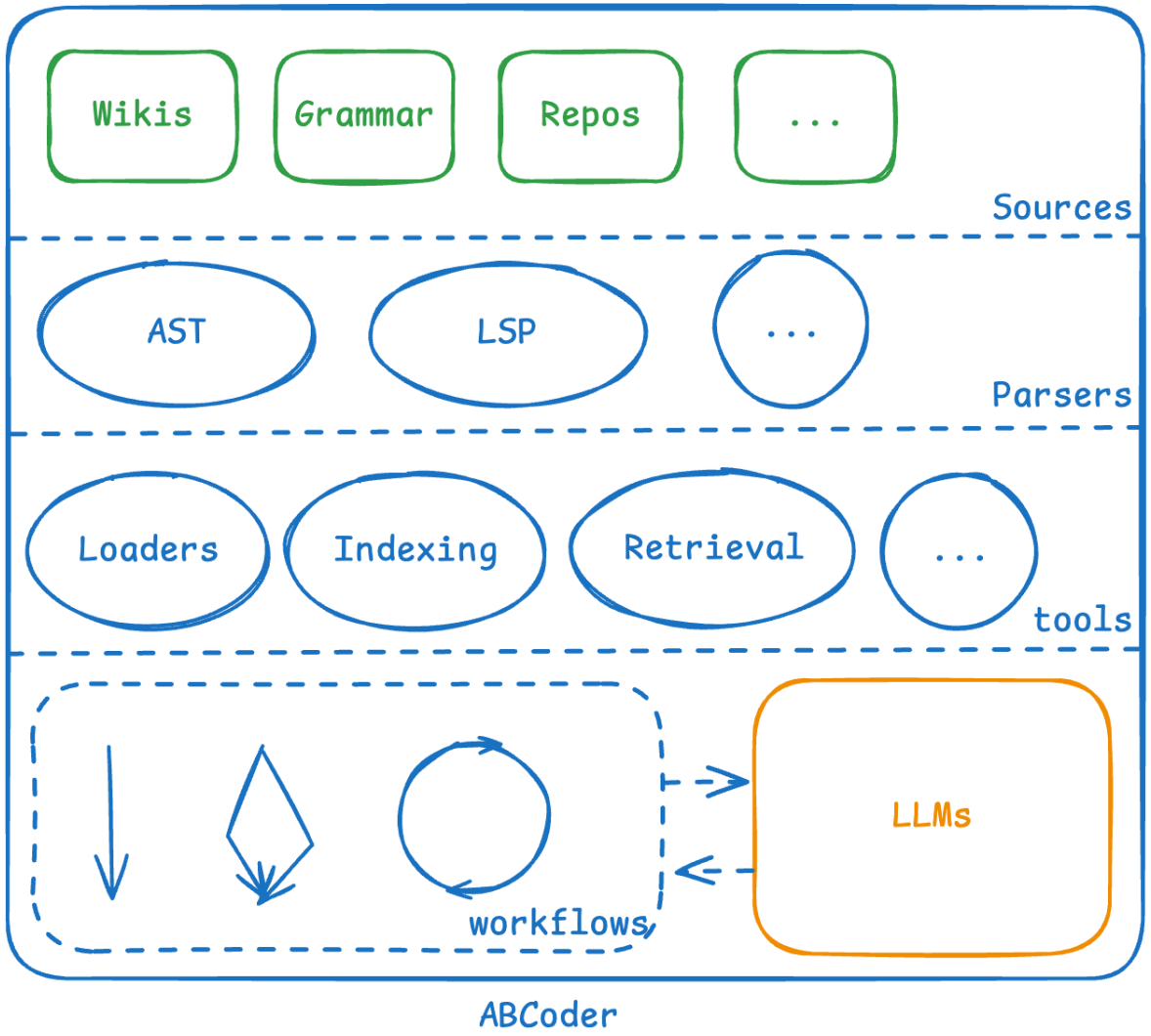
CONTENT
目录

01 缘起：ABCoder

AI-Based Coder: 基于 AI 的编程伙伴

ABCoder

ABCoder: AI-Based Coder



核心抽象：Parser

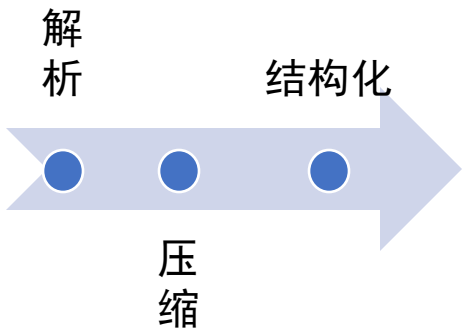
- 语言无关的语法树(AST)
- 多语言 Parser

编程落地实践：workflows

Golang Parser 实现

```
.
├── README.md
├── biz
|   ├── handler
|   |   └── ping.go
|   └── router
|       └── register.go
├── build.sh
├── go.mod
├── go.sum
├── main.go
├── router.go
├── router_gen.go
└── script
    └── bootstrap.sh
```

一个典型 Hertz 项目的 Layout




```
main
|
├── register
|   ├── router.GeneratedRegister
|   └── customizedRegister
|       └── handler.Ping
|
└── server.Default # thirdparty deps
```

可以把一个Go项目，以 main()函数为根节点，以调用关系组成一颗调用树。树中每一个节点记录着源码信息，调用关系，被调关系。

基于每个节点(主要是 func)的记录的内容，深度优先遍历每一个节点，由于每个节点都包含了调用关系，所以可以更全面地总结该节点的作用、实现过程

Golang Parser 实现

```
main
|
├── register
|   ├── router.GeneratedRegister
|   └── customizedRegister
|       └── handler.Ping
|
└── server.Default # thirdparty deps
```

1. **handler.Ping**: “这是一个处理/ping路由的函数。它返回一个pong的 JSON 响应，表示服务正常。”
2. **customizedRegister**: “该函数用于注册自定义的路由。在目前的实现中，它注册了一个/ping路由，处理函数为handler.Ping。”
3. **router.GeneratedRegister**: “这是一个占位函数，通常用于将生成的路由注册到 Hertz 服务器。目前它没有实际的代码实现，通常是由生成工具填充代码。”
4. **register**: “该函数负责注册所有的路由。它首先调用router.GeneratedRegister(r)注册生成的路由，然后调用customizedRegister(r)注册自定义路由。”
5. **main**: “该函数是应用程序的入口点。它创建一个默认的 Hertz 服务器实例，然后调用register函数注册路由，最后启动服务器并开始监听请求。”


```
"main": {
    "Exported": false,
    "IsMethod": false,
    "ModPath": "github.com/cloudwego/biz-demo/easy_note",
    "PkgPath": "github.com/cloudwego/biz-demo/easy_note/cmd/api",
    "Name": "main",
    "File": "main.go",
    "Line": 41,
    "Content": "func main() {\n\th := server.Default()\n\n\tregister(h)\n\th.Spin()\n}",
    "FunctionCalls": [
        {
            "ModPath": "github.com/cloudwego/biz-demo/easy_note",
            "PkgPath": "github.com/cloudwego/biz-demo/easy_note/cmd/api",
            "Name": "register"
        }
    ],
    "MethodCalls": [
        {
            "ModPath": "github.com/cloudwego/hertz@v0.9.1",
            "PkgPath": "github.com/cloudwego/hertz/pkg/app/server",
            "Name": "server.Default"
        }
    ],
    "GolobalVars": [],
    "compress_data": "该函数是应用程序的入口点。它创建一个默认的 Hertz 服务器实例，
                      然后调用register函数注册路由，最后启动服务器并开始监听请求。"
}
```


语言无关的 Schema 抽象

Source code as Knowledge

应用实践

应用名称	描述	核心 Tools & Workflows	知识库依赖
项目理解	ABCoder 能力最直接的应用落地：项目高质量注释和文档生成	1. ABCoder Repo Loaders 2. 文件系统 Tools	1. ABCoder 解析 & 级联压缩后仓库语料
研发辅助	ABCoder 综合应用：具备CloudWeGo 项目源码级知识和运用能力（许愿式开发）： 提出需求->设计->自动化开发->测试-> 上线	同上，额外： 1. ABCoder Parsers 2. IDL tools 3. CMD tools	同上，额外： 1. Examples as Knowledge 2. CloudWeGo 用户文档
语言翻译	多系统协作联动应用：在 ABCoder 能力之上构建业务导向能力： 半空：渐进式 Go2Rust 迁移工作流	同上，额外： 1. 复合子系统（IDE 协同、人机协同等）	同上，额外： 1. 更丰富的语言映射知识库
...			

02 半空: 渐进式 Go2Rust 工作流

快速掌握 Rust，轻松开启高效、安全的编程之旅

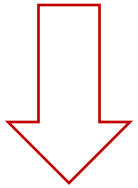
背景

Rust 性能收益

- 公司内部分 Golang 服务在迁移至 Rust 后，取得明显的性能收益 & 维护成本降低
- 公司内大量 Golang 服务消耗大头资源，期望将其迁移到 Rust，取得性能收益

Rust 落地难度

- Rust 上手难度高，熟练使用 Rust 的研发人员比较少
- Rust 生态没有 Golang 丰富，需要建设对应生态



基于 ABCoder 的代码深度理解能力，去做一套 LLM 驱动的 Golang 自动迁移至 Rust 的方案，并且让研发同学在迁移过程中掌握 Rust 的使用

->

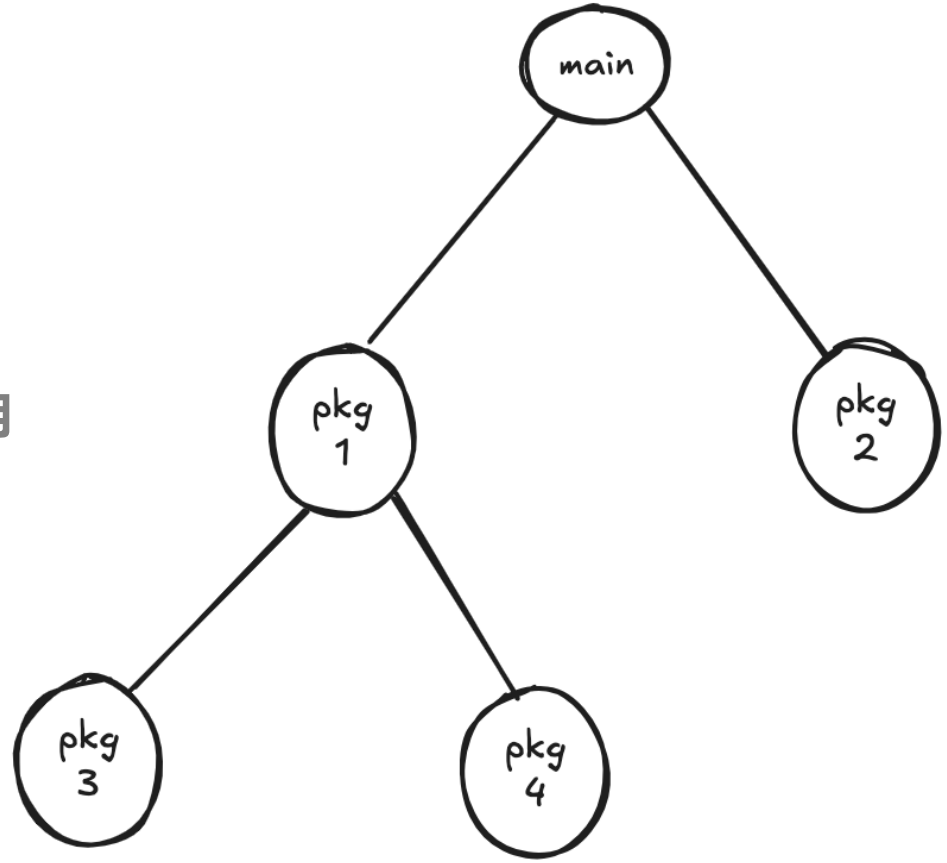
「半空」：渐进式 Go2Rust 工作流

渐进式翻译

定义

- 不会将整个项目全部翻译完再交付给用户

- 以模块层级的形式，逐个模块去翻译，逐个交付给用户，用户可随时停止翻译
  - main
  - pkg1
  - pkg2
  - pkg3
  - pkg4



为什么要渐进式翻译?

整体交付一个项目会有什么问题?

- **代码可读性差**：大量的 Rust 代码导致用户无从下手，无法充分地与原 Golang 代码进行业务逻辑正确性的对比

- **面临海量报错，修复难度大意愿低**：LLM 翻译后的 Rust 代码无法完全保证可编译，用户可能面临海量报错，迁移意愿低

- **翻译错误传播**：某一处逻辑翻译错了，可能会导致与其关联的调用都发生错误，无法保证翻译后的项目的正确性

为什么要渐进式翻译?

渐进式翻译的优势

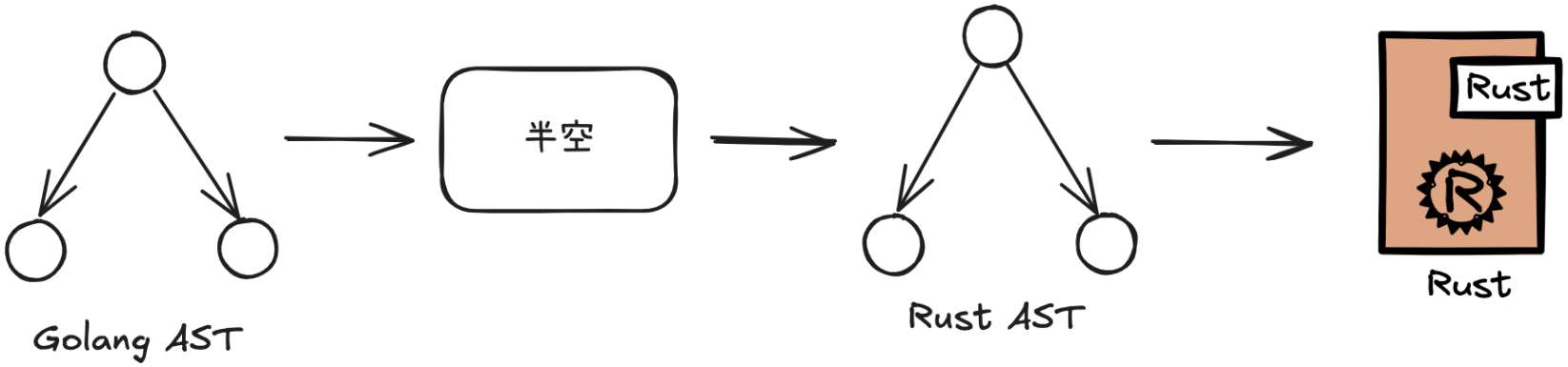
- **交付代码量少**：确保用户有足够的精力去检查生成代码的逻辑正确性

- **随时启动、随时停止，进度可控:** 当翻译到某一轮 Package 后，用户可停止使用半空辅助翻译，自行开始手动项目翻译

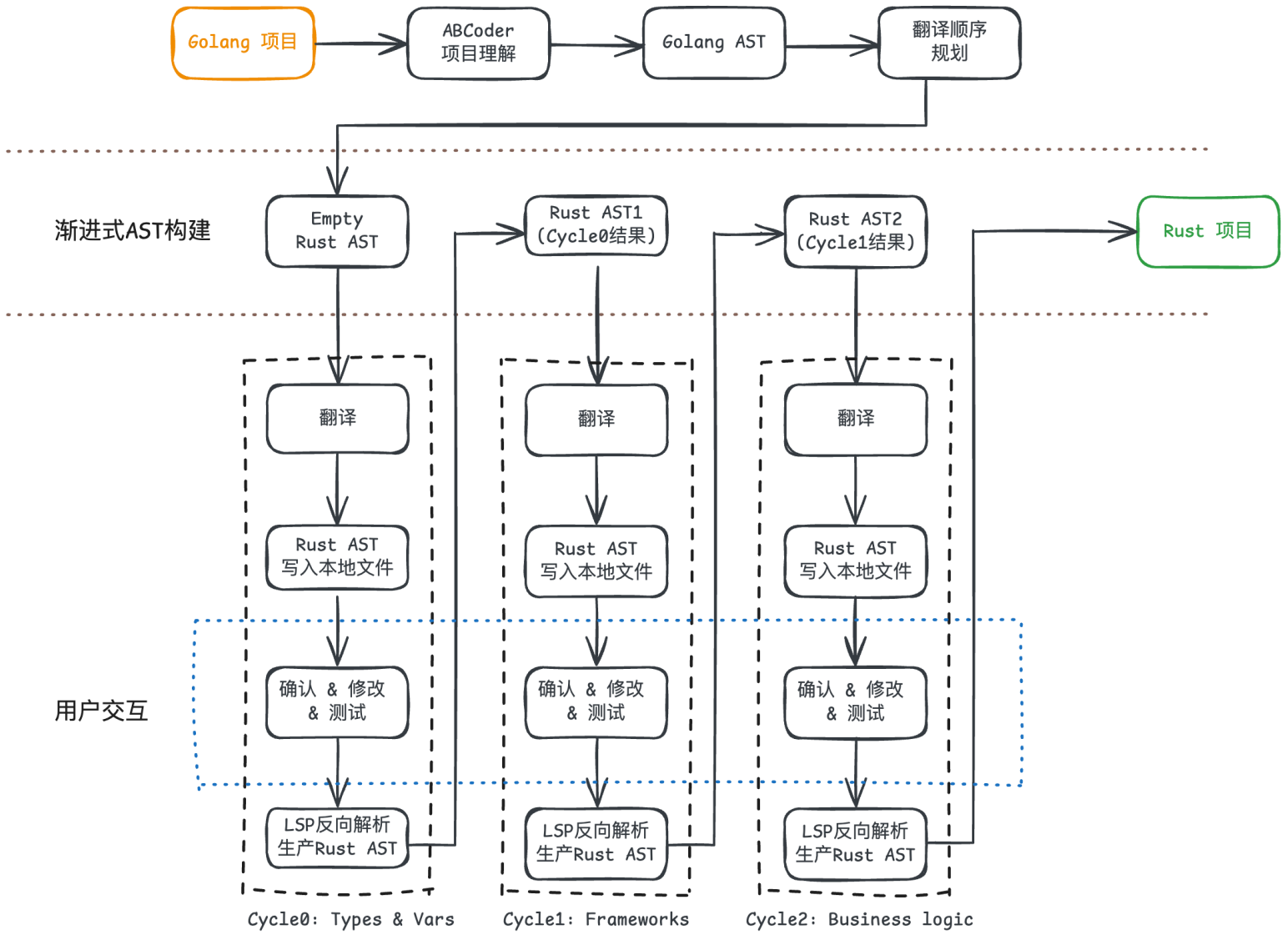
- **人工交互**：每轮 Package 翻译都需要人工校准，确保每一轮翻译的正确性
  - 通过 “不理解 -> 搜索 -> 理解” 的过程，这样的思维模式可以帮助用户快速掌握 Rust 的基本语法
  - 通过每轮的校验，也能让用户对项目的流程、细节有更深刻的印象

翻译目标

- Golang AST 经过半空系统翻译为 Rust AST
- Rust AST 通过 Generator 写回 Rust 源码



半空：渐进式翻译过程



1. ABCoder 项目理解
2. Package 翻译顺序规划
3. 按照 WorkFlow 翻译，渐进式构建 Rust AST
4. 生成 Rust 项目、用户修改 & 确认

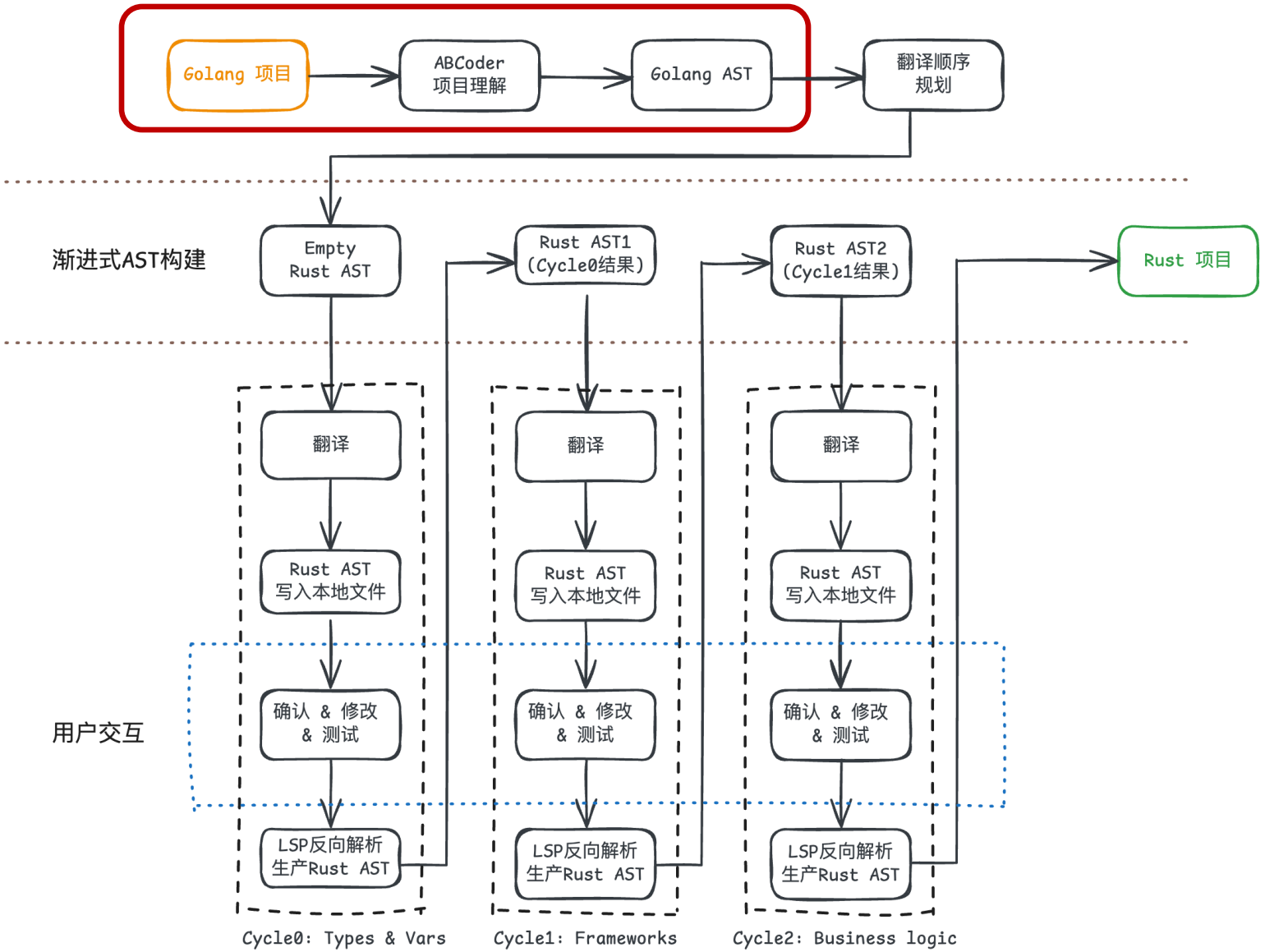
1. **ABCoder 项目理解**

2. Package 翻译顺序规划

3. 按照 WorkFlow 翻译，渐进式构建 Rust AST

4. 生成 Rust 项目、用户修改 & 确认

ABCoder 项目理解



- 完成上一节 ABCoder 解析过程
- 获取一颗完整的项目理解 AST

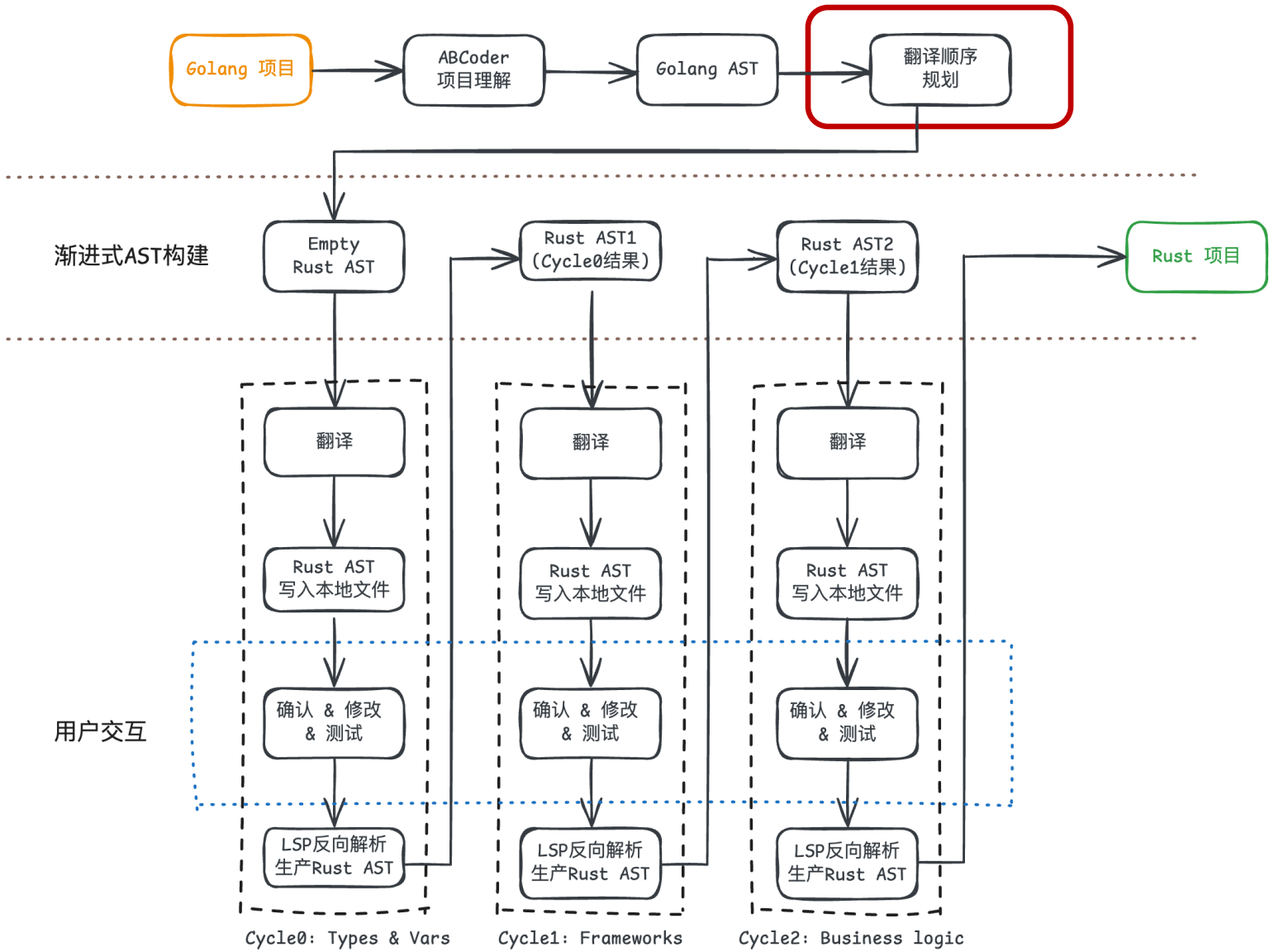
1. ABCoder 项目理解

2. Package 翻译顺序规划

3. 按照 WorkFlow 翻译，渐进式构建 Rust AST

4. 生成 Rust 项目、用户修改 & 确认

Package 翻译顺序规划



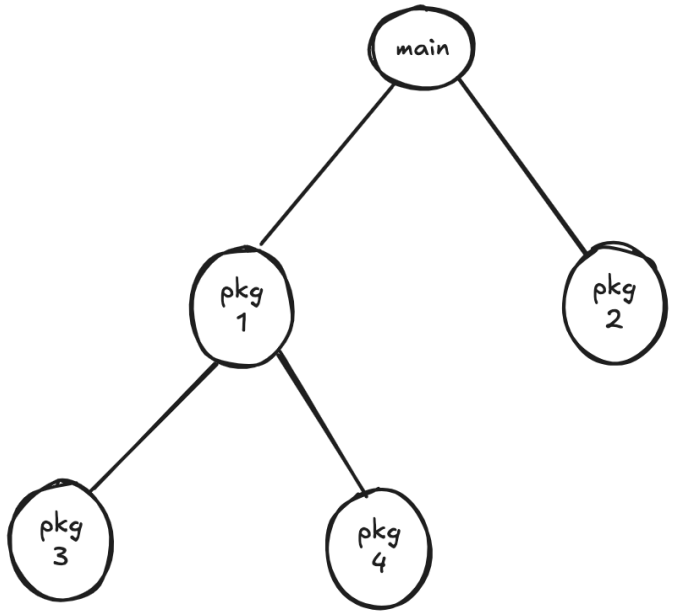
- 构建 Package 依赖关系图
- 按照 Package 自顶向下的顺序展开，被依赖少的 Package 优先翻译

Package 翻译顺序规划

自顶向下 VS 自底向上？

	自顶向下	自底向上
翻译顺序	• Main Package 优先 • 自顶向下层序遍历展开	• 无其他依赖的函数优先；(common、util) • 自底向上层序遍历展开
优势	• 符合人写代码的习惯，先搭建出项目骨架，再不断完善 • 尽量确保每一轮结果都可以编译、运行 • 可随时停止，人工续写其他代码	• 可优先实现无依赖的 Package 中的函数，便于被其他函数调用
劣势	• 函数中调用的其他函数可能还没有实现，无法编译 -> 通过先mock再实现的方式解决	• 整体项目要到最后才能开启编译 • 每轮翻译结果对于用户无从下手

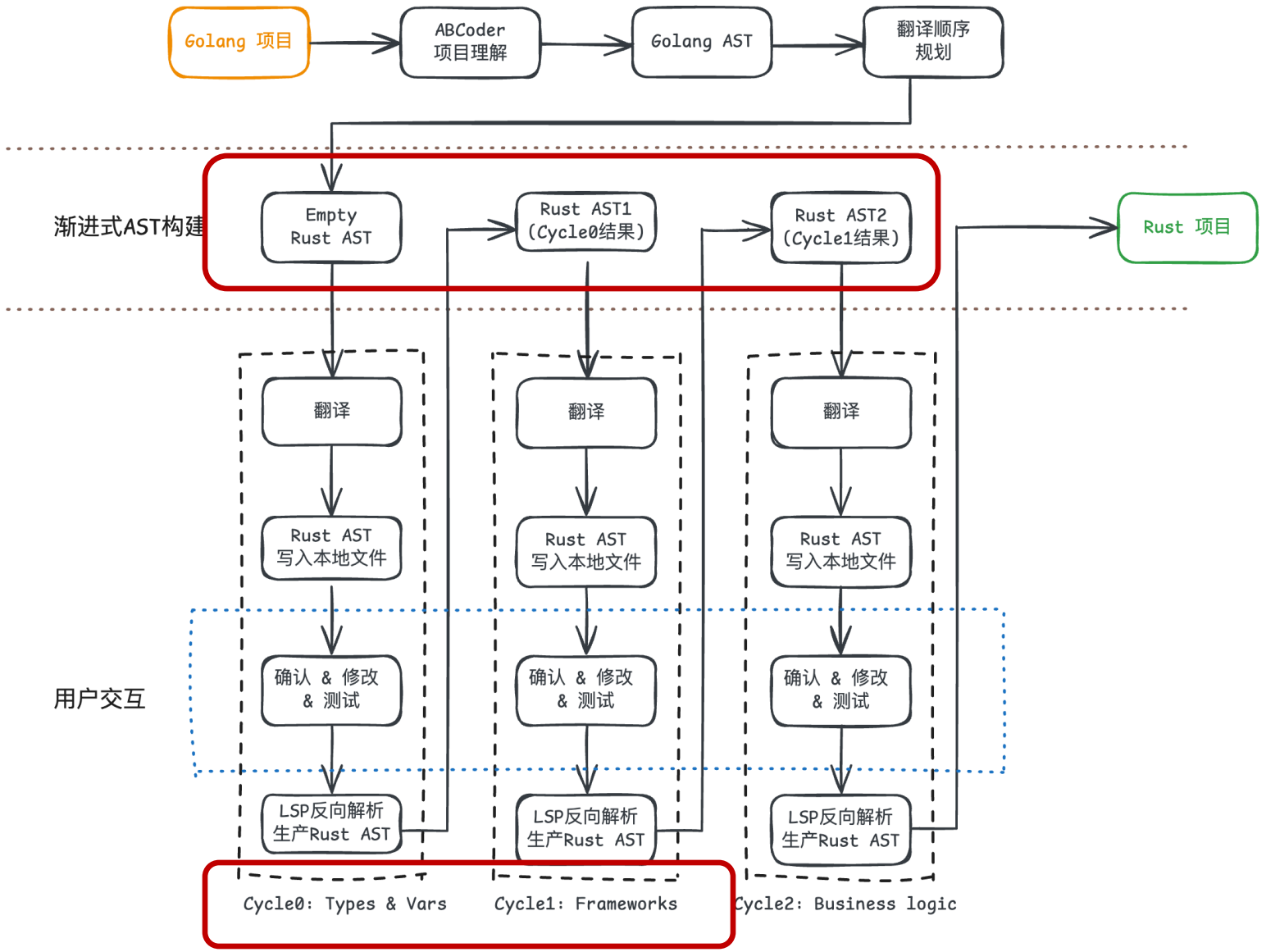
- Cycle0 type&var：按照 Package 自底向上
- 正式翻译：按照 Package 自顶向下



自顶向下顺序：main、pkg1、pkg2、pkg3、pkg4

自底向上顺序：pkg3、pkg4、pkg1、pkg2、main

Package 翻译顺序规划



1. Cycle0: 框架映射
   - 天然适配 CloudWeGo 开源的各种框架
   - Kitex/Hertz 框架会自动映射为 Volo/Volo-HTTP 框架，并完成对应代码生成

2. Cycle0: 类型定义&全局变量翻译
   - 类型和变量的定义相对独立，翻译难度低
   - 提前翻译好，可供后续翻译直接使用
   - 翻译过程完成相关生命周期、所有权的设计(并不会100%准确)。

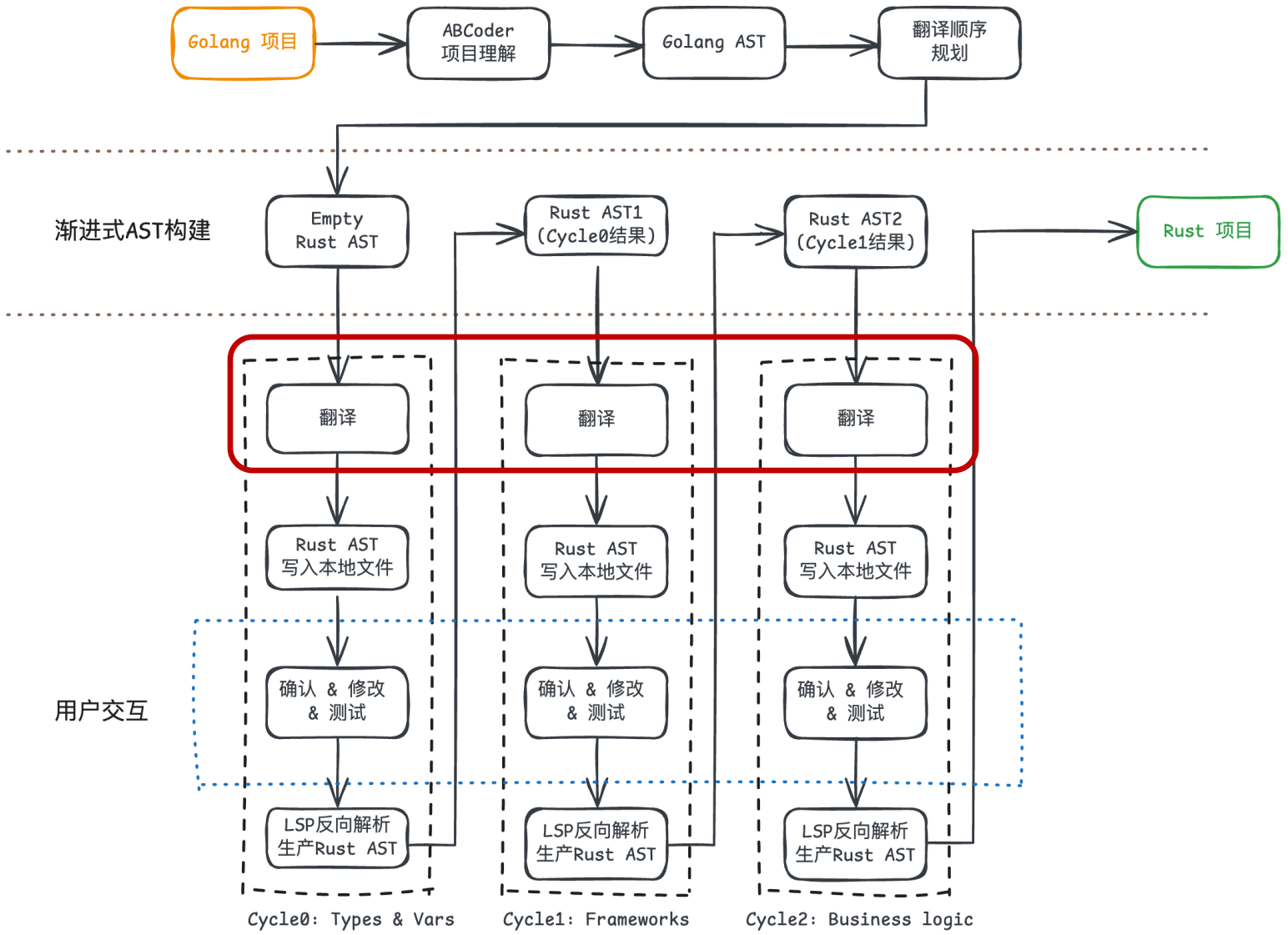
1. **ABCoder 项目理解**

2. Package 翻译顺序规划

3. 按照 WorkFlow 翻译，渐进式构建 Rust AST

4. 生成 Rust 项目、用户修改 & 确认

翻译WorkFlow



- 基于半空的翻译 WorkFlow 进行翻译
- WorkFlow 会对当前 Package 每个节点进行逐个翻译
- 翻译过程的本质就是将当前 Package 中的节点翻译成 Rust，并插入到 Rust AST 中

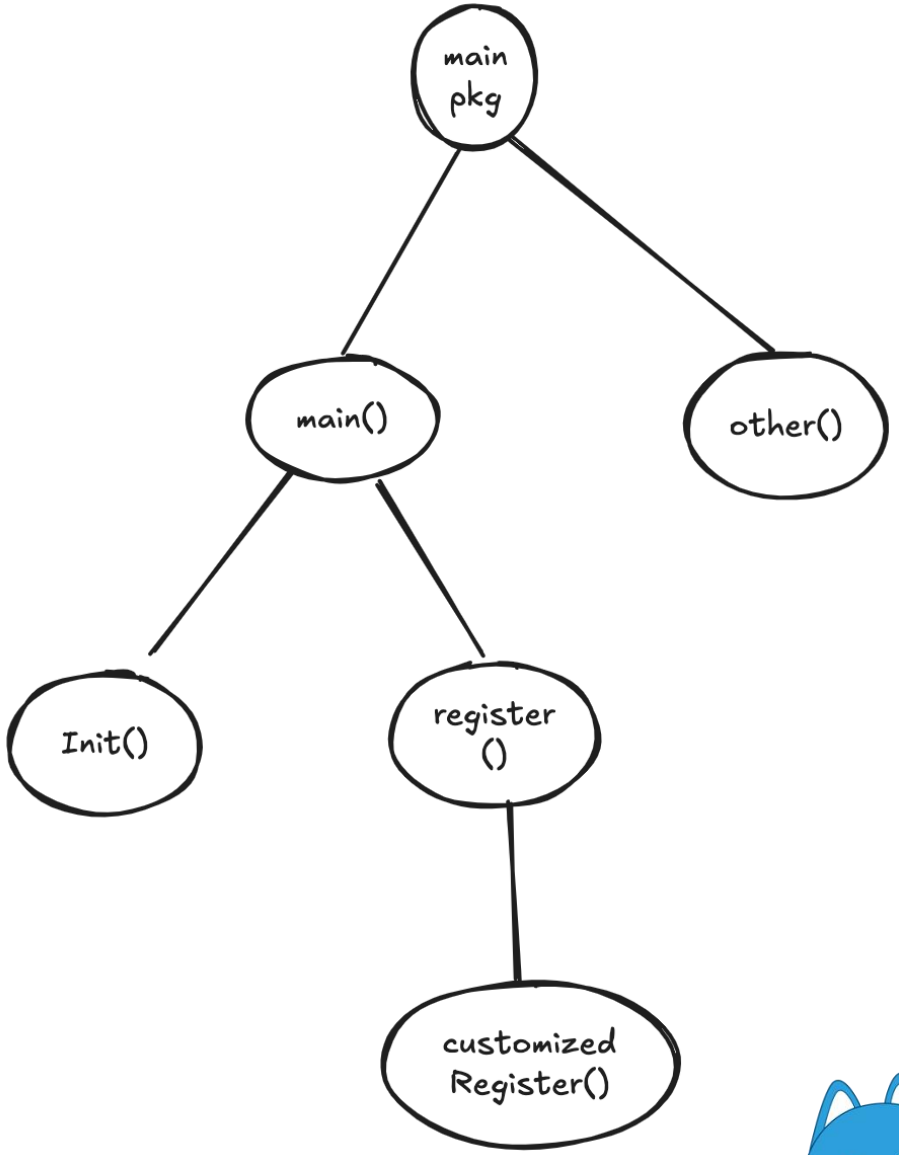
翻译 WorkFlow

Package 内函数节点的翻译顺序

- 一个 Package 下会有若干函数/方法定义
- 函数/方法有的会相互调用，有的则是完全独立
- 使用拓扑排序，优先翻译被依赖少的节点

翻译顺序如下

1. main()
2. other()
3. Init()
4. register()
5. customizedRegister()



easy_note_api

翻译 WorkFlow


```
"main": {
    "Exported": false,
    "IsMethod": false,
    "ModPath": "github.com/cloudwego/biz-demo/easy_note",
    "PkgPath": "github.com/cloudwego/biz-demo/easy_note/cmd/api",
    "Name": "main",
    "File": "main.go",
    "Line": 41,
    "Content": "func main() {}",
    "FunctionCalls": [
        {
            "ModPath": "github.com/kitex-contrib/obs-opentelemetry@v0.2.7",
            "PkgPath": "github.com/kitex-contrib/obs-opentelemetry/provider",
            "Name": "NewOpenTelemetryProvider"
        },
        {
            "ModPath": "github.com/cloudwego/biz-demo/easy_note",
            "PkgPath": "github.com/cloudwego/biz-demo/easy_note/cmd/api",
            "Name": "Init"
        },
    ],
    "MethodCalls": […
    ],
    "GolobalVars": […
    ],
    "compress_data": "
    main函数的主要功能是初始化和启动一个基于Hertz的HTTP服务器，其中集成了OpenTelemetry和pprof中
    1. 创建OpenTelemetryProvider实例，配置服务名、导出端点和非安全连接。
    2. 使用defer确保在程序结束时关闭OpenTelemetryProvider。
    3. 调用Init函数初始化应用程序组件，如RPC、JWT中间件和日志系统。
    4. 创建ServerTracer实例用于跟踪。
    5. 创建并配置Hertz服务器，设置监听端口和处理方法不允许等选项。
    6. 注册pprof中间件。
    7. 使用OpenTelemetry中间件。
    8. 注册HTTP路由配置。
    9. 启动服务器进行监听和处理请求。"
}
```


任务规划
LLM Agent

生成语言无关的需求描述


```
{
  "target": "main函数的主要功能是初始化和启动一个基于Hertz的HTTP服务器，其中集成了OpenTelemetry和pprof中间件。其具体步骤
    在程序结束时关闭OpenTelemetryProvider。\n3. 调用Init函数初始化应用程序组件，如RPC、JWT中间件和日志系统。\n4. 创建Se
    pprof中间件。\n7. 使用OpenTelemetry中间件。\n8. 注册HTTP路由配置。\n9. 启动服务器进行监听和处理请求。",
  "steps": [
    { 1
      "detail": "创建一个OpenTelemetryProvider实例，配置服务名、导出端点和非安全连接。",
      "dependencies": […
      ]
    },
    { 2
      "detail": "使用defer确保在程序结束时关闭OpenTelemetryProvider。",
      "dependencies": […
      ]
    },
    { 3
      "detail": "调用Init函数初始化应用程序组件，如RPC、JWT中间件和日志系统。",
      "dependencies": […
      ]
    },
    { 4
      "detail": "创建ServerTracer实例用于跟踪。",
      "dependencies": […
      ]
    },
    { 5
      "detail": "创建并配置Hertz服务器，设置监听端口和处理方法不允许等选项。",
      "dependencies": […
      ]
    },
    { 6
      "detail": "注册pprof中间件。",
      "dependencies": […
      ]
    },
    { 7
      "detail": "使用OpenTelemetry中间件。",
      "codes": "h.Use(tracing.ServerMiddleware(cfg))",
      "dependencies": […
      ]
```

翻译 WorkFlow


```
{
  "target": "main函数的主要功能是初始化和启动一个基于Hertz的HTTP服务器，其中集成了OpenTelemetry和pprof中间件。其具体步骤
    在程序结束时关闭OpenTelemetryProvider。\n3. 调用Init函数初始化应用程序组件，如RPC、JWT中间件和日志系统。\n4. 创建Se
    pprof中间件。\n7. 使用OpenTelemetry中间件。\n8. 注册HTTP路由配置。\n9. 启动服务器进行监听和处理请求。",
  "steps": [
    {
      "detail": "创建一个OpenTelemetryProvider实例，配置服务名、导出端点和非安全连接。",
      "dependencies": […
      ]
    },
    {
      "detail": "使用defer确保在程序结束时关闭OpenTelemetryProvider。",
      "dependencies": […
      ]
    },
    {
      "detail": "调用Init函数初始化应用程序组件，如RPC、JWT中间件和日志系统。",
      "dependencies": […
      ]
    },
    {
      "detail": "创建ServerTracer实例用于跟踪。",
      "dependencies": […
      ]
    },
    {
      "detail": "创建并配置Hertz服务器，设置监听端口和处理方法不允许等选项。",
      "dependencies": […
      ]
    },
    {
      "detail": "注册pprof中间件。",
      "dependencies": […
      ]
    },
    {
      "detail": "使用OpenTelemetry中间件。",
      "codes": "h.Use(tracing.ServerMiddleware(cfg))",
      "dependencies": […
      ]
```


知识召回
LLM Agent

根据需求召回 Rust 知识


```
{
  "HTTP Server": [ 1
  ],
  "HTTP handler": [ 2
  ],
  "HTTP 框架": [ 3
    "# HTTP 框架\n请使用 lust-http 框架作为基础的 rust http 框架.\n"
  ],
  "HTTP 路由": [ 4
    "# HTTP 路由\nlust-http 路由的可以使用 get/post/any/delete/patch/h
      index_handler() -\u003e \u0026'static str {\n    Router::ne
  ],
  "OpenTelemetry": [ 5
  ],
  "Pprof 性能分析": [ 6
  ],
  "当前包的总体作用": [ 7
  ],
  "链路追踪/日志": [ 8
  ]
}
```

翻译 WorkFlow


```
{
  "detail": "调用Init函数初始化应用程序组件，如RPC、JWT中间件和日志系统。",
  "dependencies": [
    {
      "id": "github.com/cloudwego/biz-demo/easy_note/cmd/api#Init",
      "description": "初始化应用程序的各种组件，包括RPC、JWT中间件和日志系统。"
    }
  ]
},
```

静态映射

```
{
  "detail": "调用Init函数初始化应用程序组件，如RPC、JWT中间件和日志系统。",
  "dependencies": [
    {
      "id": "crate::cmd::api#init",
      "description": "初始化应用程序的各种组件，包括RPC、JWT中间件和日志系统。"
    }
  ]
},
```

前置MOCK
LLM Agent

所有的依赖函数Mock

```rust
// mock for implement stage
pub fn init() {
    todo!("implement it");
}
```

翻译 WorkFlow


```
{
  "id": "crate::cmd::api#main",   目标 Rust ID
  "type": "FUNC",
  "target": "main函数的主要功能是初始化和启动一个基于lust-http的HTTP服务器，其中集成了OpenTelemetry和pprof中间
      defer确保在程序结束时关闭OpenTelemetryProvider。\n3. 调用Init函数初始化应用程序组件，如RPC、JWT中间件和日
      等选项。\n6. 注册pprof中间件。\n7. 使用OpenTelemetry中间件。\n8. 注册HTTP路由配置。\n9. 启动服务器进行监
  "steps": [   实现过程
      {
          "detail": "调用Init函数初始化应用程序组件，如RPC、JWT中间件和日志系统。",
          "dependencies": [
              {
                  "id": "crate::cmd::api#init",   该过程调用函数的引用路径
                  "description": "初始化应用程序的各种组件，包括RPC、JWT中间件和日志系统。",
                  "content": "// mock for implement stage\npub fn init() {\n    todo!(\"implement it
              }
          ]
      }
  ],
  "knowledges": {   使用到的Rust知识
      "HTTP 框架": [
      ],
      "HTTP 路由": [
      ]
  }
}
```

目标任务

最终实现
LLM Agent

完成最终Rust翻译

```rust
#[volo::main]
async fn main() {
    // 1. 创建OpenTelemetry提供者，配置服务名称、导出端点和安全设置
    // let tracer = opentelemetry_otlp::new_pipeline()
    //     .tracing()
    //     .with_endpoint(EXPORT_ENDPOINT)
    //     .with_trace_config(
    //         sdktrace::config().with_resource(Resource::new(vec![
    //             KeyValue::new("service.name", API_SERVICE_NAME),
    //         ])),
    //     )
    //     .install_simple()
    //     .expect("Failed to install OpenTelemetry tracer");

    // let opentelemetry = OpenTelemetryLayer::new(tracer);
    // let subscriber = Registry::default().with(opentelemetry);
    // tracing::subscriber::set_global_default(subscriber).expect("Failed

    // 2. 注册一个延迟关闭操作以确保OpenTelemetry提供者在程序结束时正确关闭
    // (Note: This is a pseudo-code to indicate the intention)
    // let _guard = opentelemetry::global::shutdown_tracer_provider();

    // 3. 初始化系统组件
    init();

    // 4. 创建一个新的服务器实例，配置端口、处理方法和追踪器
    let app = Router::new()
        // .merge(pprof()) // 5. 注册 pprof 中间件
        .merge(register()); // 6. 在服务器上注册路由和处理器

    let addr = "[::]:8080".parse::<SocketAddr>().unwrap();
    let addr = Address::from(addr);

    println!("Listening on {addr}");

    // 7. 启动服务器
    Server::new(app).run(addr).await.unwrap();
}
```

半空自动将未有的依赖注释掉

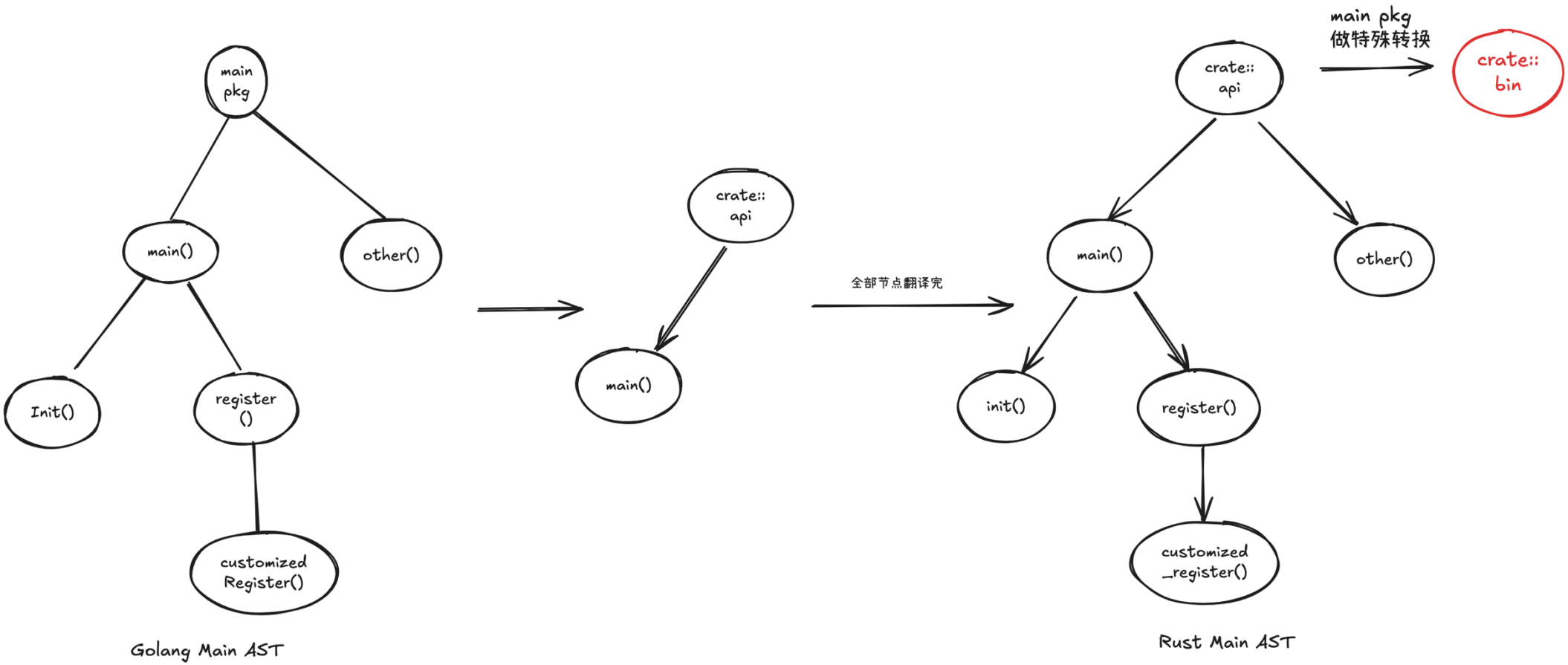
翻译 WorkFlow


```go
func main() {
	p := provider.NewOpenTelemetryProvider(
		provider.WithServiceName(consts.ApiServiceName),
		provider.WithExportEndpoint(consts.ExportEndpoint),
		provider.WithInsecure(),
	)
	defer p.Shutdown(context.Background()) // nolint:errcheck

	Init()
	tracer, cfg := tracing.NewServerTracer()
	h := server.New(
		server.WithHostPorts(":8080"),
		server.WithHandleMethodNotAllowed(true), // coordinate
		tracer,
	)
	// use pprof mw
	pprof.Register(h)
	// use otel mw
	h.Use(tracing.ServerMiddleware(cfg))
	register(h)
	h.Spin()
}
```

```rust
#[volo::main]
async fn main() {
    // 1. 创建OpenTelemetry提供者，配置服务名称、导出端点和安全设置
    // let tracer = opentelemetry_otlp::new_pipeline()
    //     .tracing()
    //     .with_endpoint(EXPORT_ENDPOINT)
    //     .with_trace_config(
    //         sdktrace::config().with_resource(Resource::new(vec![
    //             KeyValue::new("service.name", API_SERVICE_NAME),
    //         ])),
    //     )
    //     .install_simple()
    //     .expect("Failed to install OpenTelemetry tracer");

    // let opentelemetry = OpenTelemetryLayer::new(tracer);
    // let subscriber = Registry::default().with(opentelemetry);
    // tracing::subscriber::set_global_default(subscriber).expect("Failed

    // 2. 注册一个延迟关闭操作以确保OpenTelemetry提供者在程序结束时正确关闭
    // (Note: This is a pseudo-code to indicate the intention)
    // let _guard = opentelemetry::global::shutdown_tracer_provider();

    // 3. 初始化系统组件
    init();

    // 4. 创建一个新的服务器实例，配置端口、处理方法和追踪器
    let app = Router::new()
        // .merge(pprof()) // 5. 注册 pprof 中间件
        .merge(register()); // 6. 在服务器上注册路由和处理器

    let addr = "[::]:8080".parse::<SocketAddr>().unwrap();
    let addr = Address::from(addr);

    println!("Listening on {addr}");

    // 7. 启动服务器
    Server::new(app).run(addr).await.unwrap();
}
```

半空自动将未有的依赖注释掉

翻译 WorkFlow

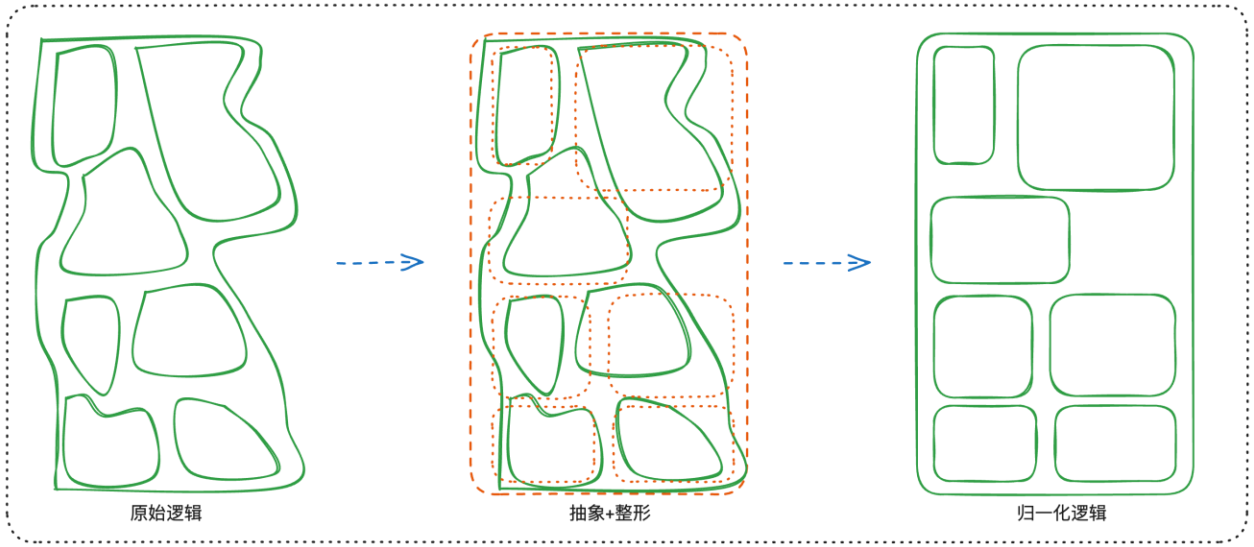

main pkg
main()
other()
Init()
register ()
customized Register()
Golang Main AST
crate:: api
main()
全部节点翻译完
crate:: api
main()
other()
init()
register()
customized _register()
Rust Main AST
main pkg
做特殊转换
crate:: bin

翻译 WorkFlow

总结

- 翻译过程是对原始 Golang 逻辑的高度抽象，在保留原始业务逻辑的同时摆脱Golang的约束；使得翻译的 Rust 代码摆脱 Golang 的影子，达到“意译”的效果

- 基于知识库可自定义“领域特定”的知识，翻译的 Rust 代码更符合相关组件的最佳实践



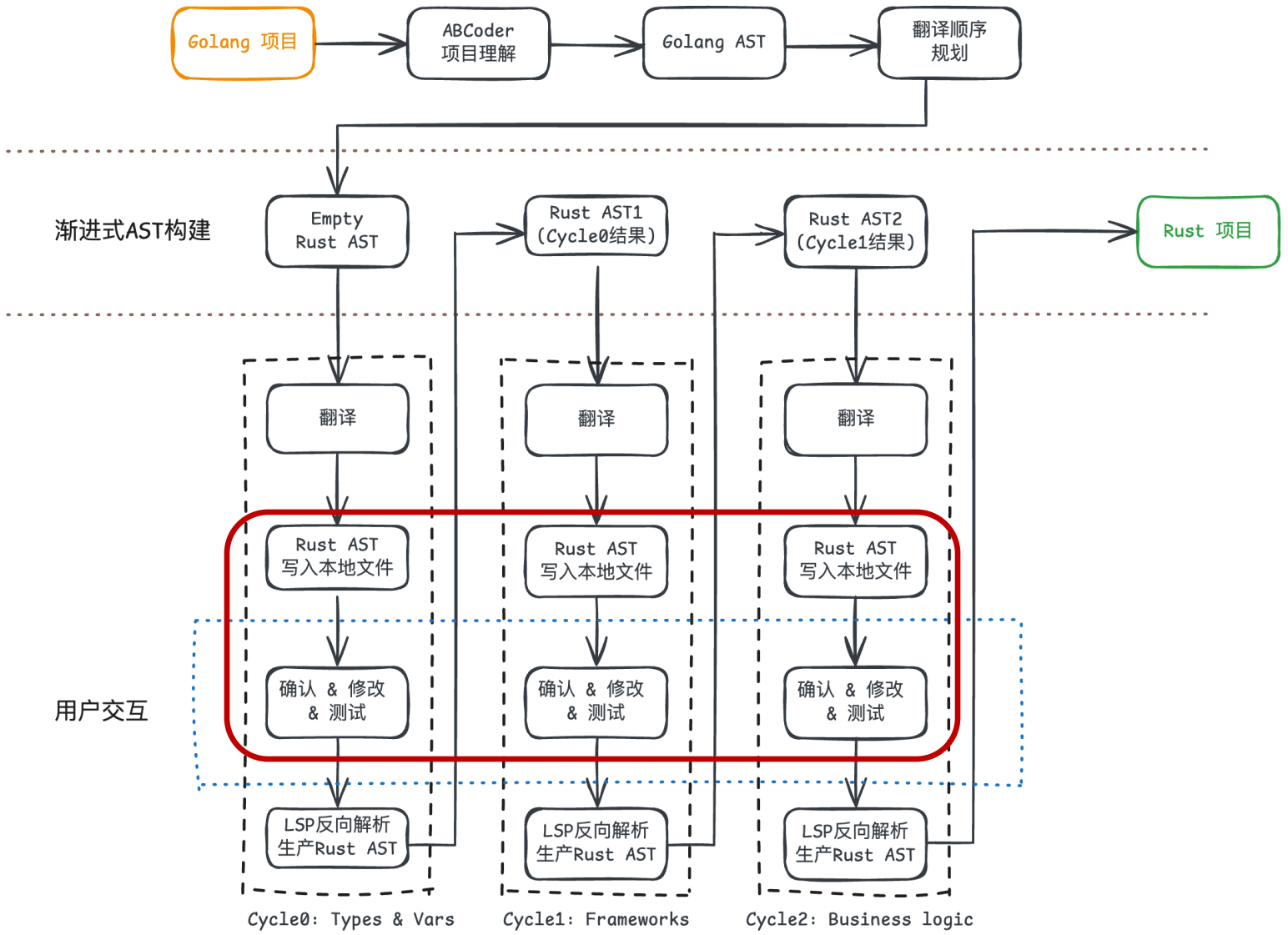
1. ABCoder 项目理解

2. Package 翻译顺序规划

3. 按照 WorkFlow 翻译，渐进式构建 Rust AST

4. 生成 Rust 项目、用户修改 & 确认

用户交互



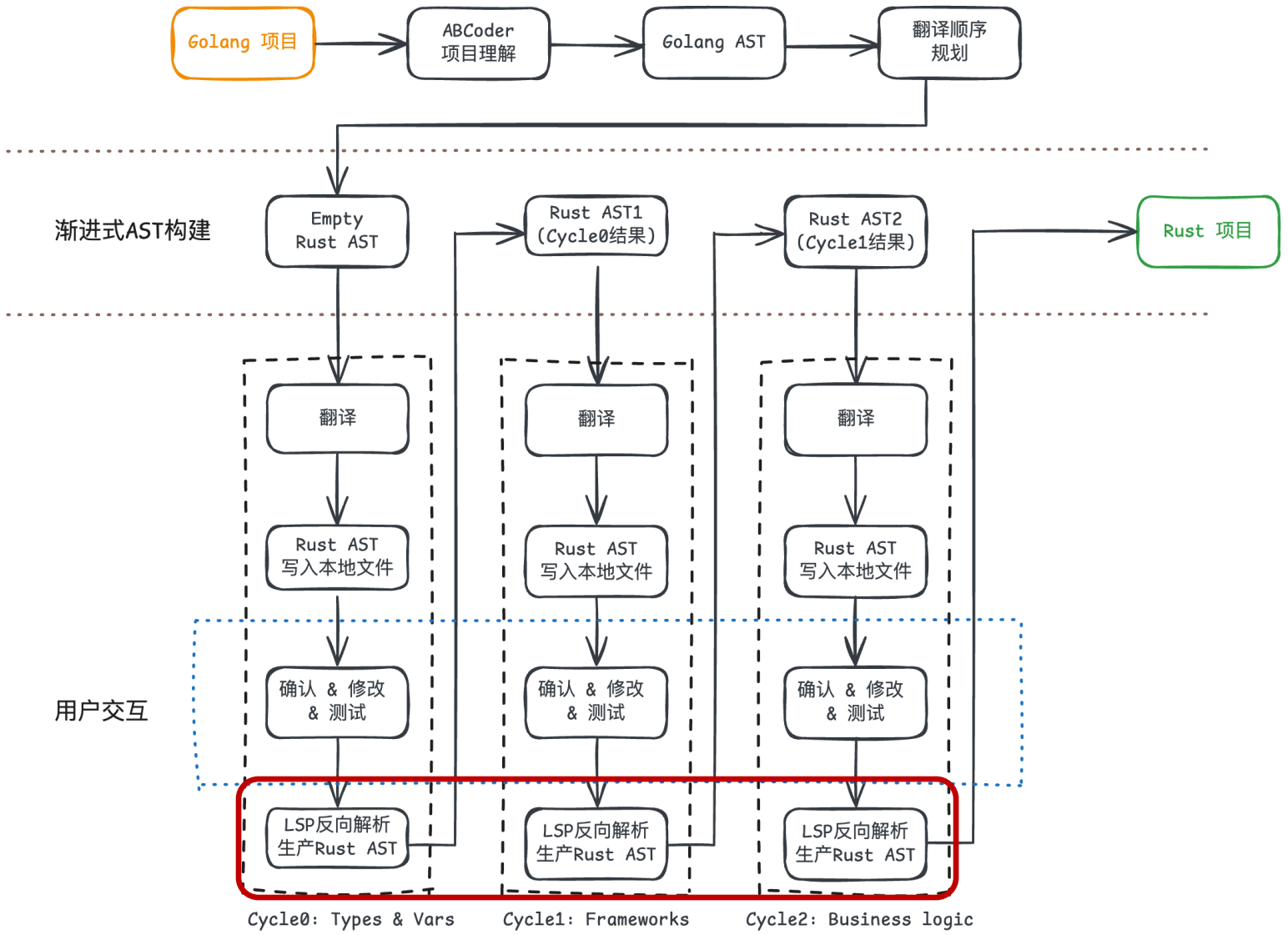
写回 Rust 文件

- 将翻译后 Rust AST 以 Rust 源码的形式写到 Rust 项目；默认使用 go Package 的文件组织形式

用户交互

- 检查是否有编译报错、逻辑错误，如有则需求修改
- 修完完毕，提交代码

用户交互



LSP 解析 Rust & 生成 AST

- 基于 LSP 开发 Rust 版本的 ABCoder 解析器
- 该解析器会解析用户的修改、新增的内容
- 记录用户修改的内容，作为用户的即时反馈
- 后续翻译以这个 Rust AST 为基础，完成其他节点的翻译，并插入对应的节点

总结：半空的特点

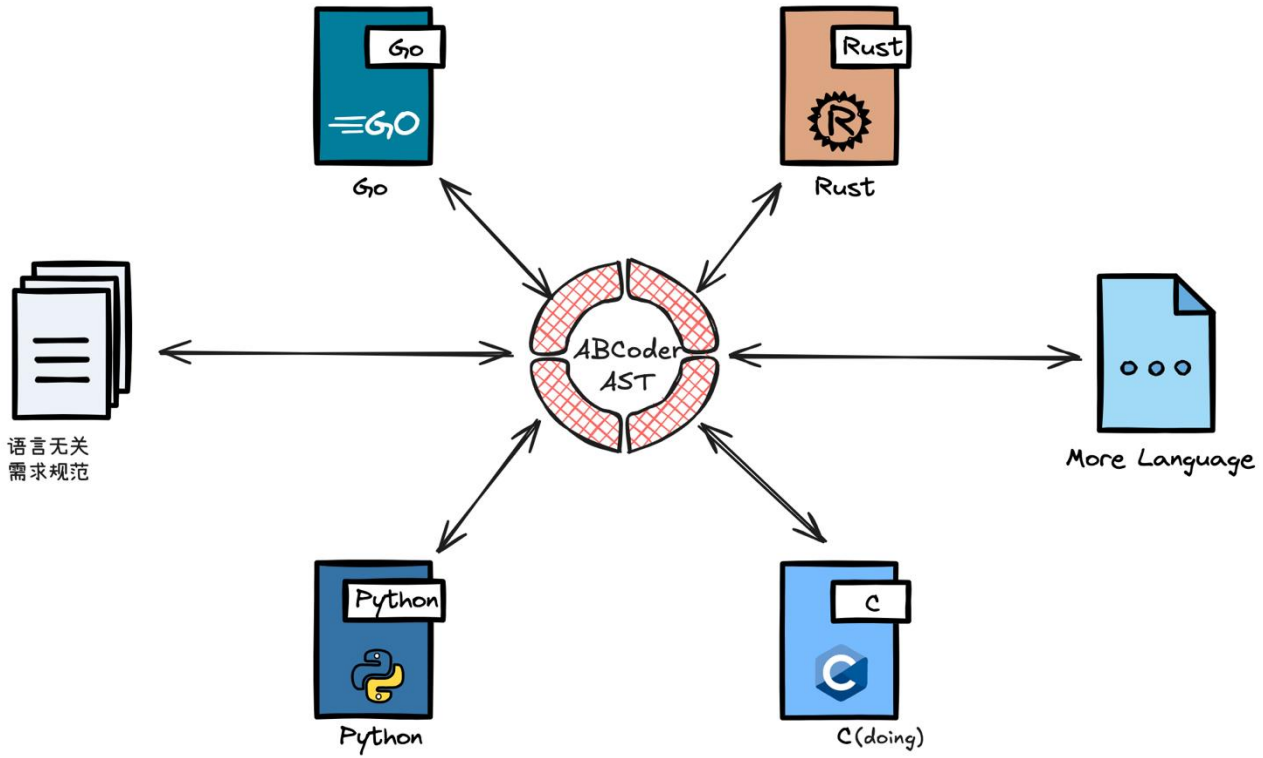
- 项目梳理：可帮用户更细致的完成项目梳理
- Rust 新手友好：可加速团队成员上手 Rust 的速度，节省人力成本
- 渐进式迁移： 基于项目设计从 0 到 1 开启迭代，进度可视、可控
- 迁移准确率高： 建立多个组件的知识库，使用各组件的最佳实践完成各个功能的迁移

03

未来展望

展望

- 更完善的人机交互能力
- 构建一个支持多语言互转的代码翻译工作流：空(Kong)



THANKS

CloudWeGo