

为什么我们用 Rust 重写了自己的项目

CNCF KCL Owner / 徐鹏飞

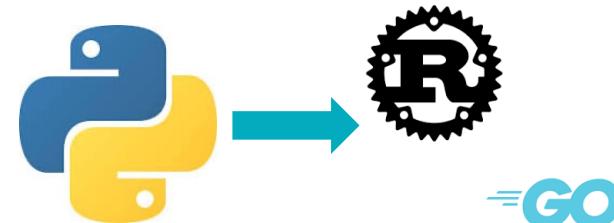
个人介绍

徐鹏飞 (GitHub: Peefy), 开源爱好者

对云原生、Serverless、区块链、AI 等领域感兴趣，目前主要维护下面几个开源项目



<https://github.com/kcl-lang>



<https://github.com/pluto-lang>



<https://github.com/AntChainOpenLabs/Smart-Intermediate-Representation>



提纲

01

重构 - 凤凰涅槃 🔥

02

演进 - 奔腾跃进 🐾

03

展望 - 星辰大海 🌌

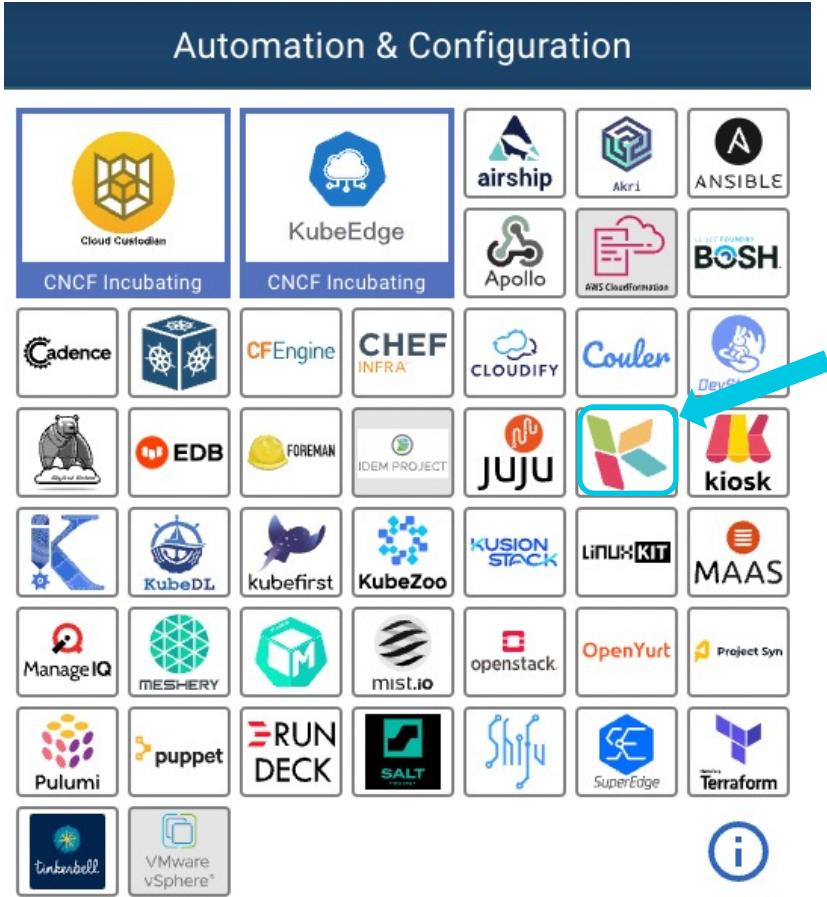
01

重构 - 洋火重生 🔥

简介 (广告时间 😊)

KCL

专用配置策略语言 (2022.6 用 Rust 重写完成后正式开源，2023.9 成为 CNCF 基金会托管的 Sandbox 项目)



- ✓ **领域特定**: 以收敛的语言和工具集合解决领域问题近乎无限的变化和复杂性，同时兼顾表达力和易用性
- ✓ **以数据和模型为中心**: 开发者可以理解的声明式 Schema/配置/策略模型用于 AI 工程，云原生工程等场景
- ✓ **包含结构化定义和约束的核心数据结构 Schema**: 为 AI 数据集或者云原生配置场景提供原生的数据验证和转换能力
- ✓ **可复用扩展**: OCI 等标准软件供应链集成和包管理工具支持，官方 Registry 提供 300+ 模型包
- ✓ **引擎解耦**: 建立在一个完全开放的世界当中，几乎不与任何编排/引擎工具或者控制器绑定，可同时为客户端和运行时场景提供 API 抽象、组合和校验的能力
- ✓ **多语言 SDK**: 轻易集成到不同的业务场景和生态当中，目前提供了 Rust, Go, Python, Java 等 SDK

项目地址: <https://github.com/kcl-lang>

当然，今天我们不聊 KCL，聊一聊和 Rust 那些事

重构

2022.1 - 2022.6 用 Rust 重写 KCL (资源投入 3 人)

- 问题
- 原有项目冷启动时间长，性能差
 - Python None NPE 问题影响稳定性，虽然测试覆盖率一度高达 90%+
 - 中型的 Python 项目维护成本，类型注解，mypy 等工具使用繁琐



50k Python Code

<https://github.com/kcl-lang/kcl-py>

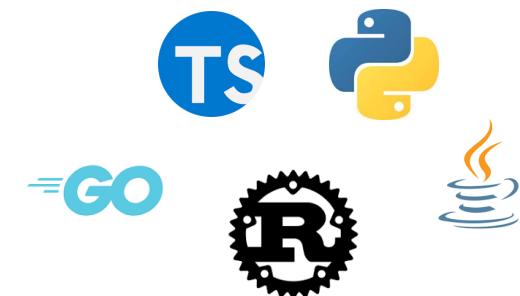
2022.1



50k Rust Code
For KCL Core

<https://github.com/kcl-lang/kcl>

2022.6

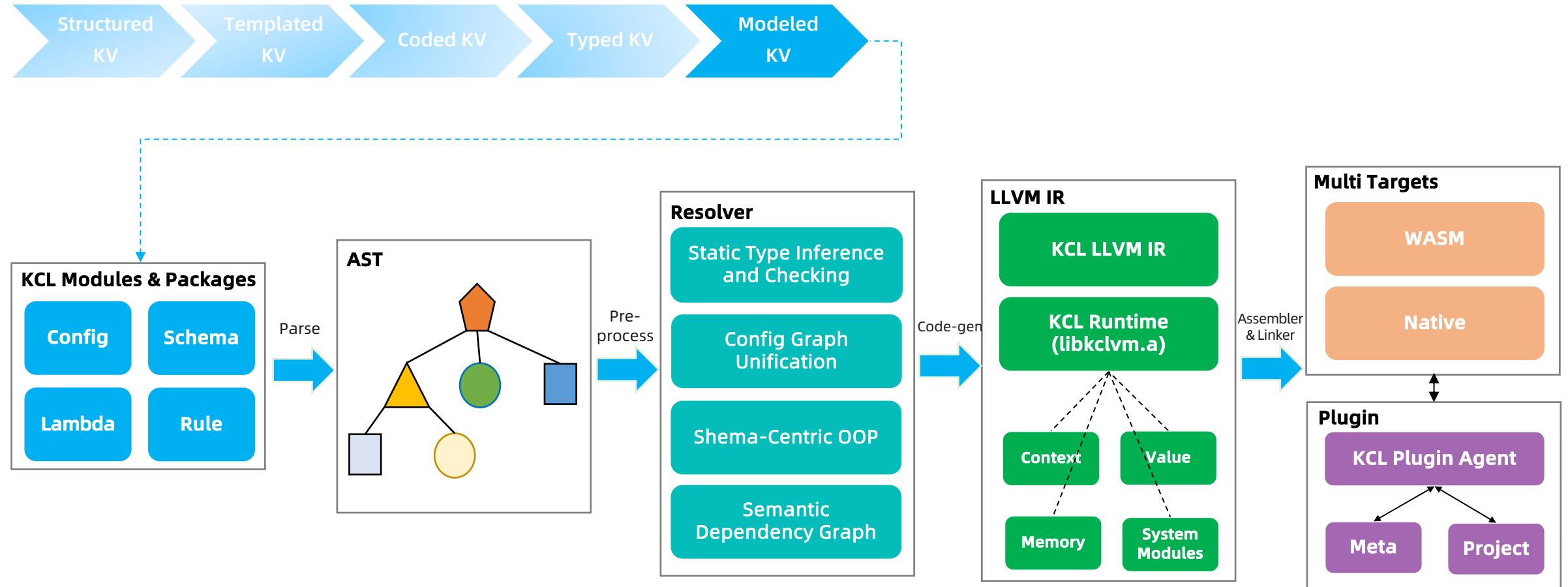


80k Rust Code
For KCL Core

Rust 核心+其他语言外围生态项目

至今

架构



分模块重写，从运行时到编译器前端

结果

综合考虑用户体验、稳定性和性能决定重写，各维度指标均有提升

66%

E2E Perf

20x

Parser

40x

Resolver

0

NPE Bugs

1/40

Cold Start

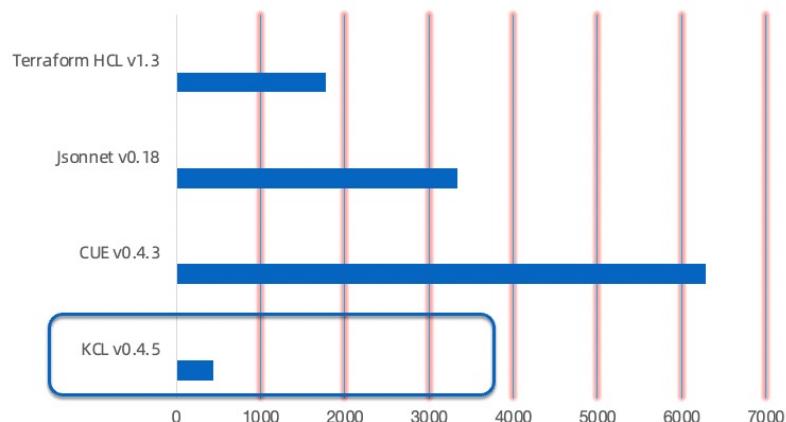
1/2

Runtime
Memory

结果

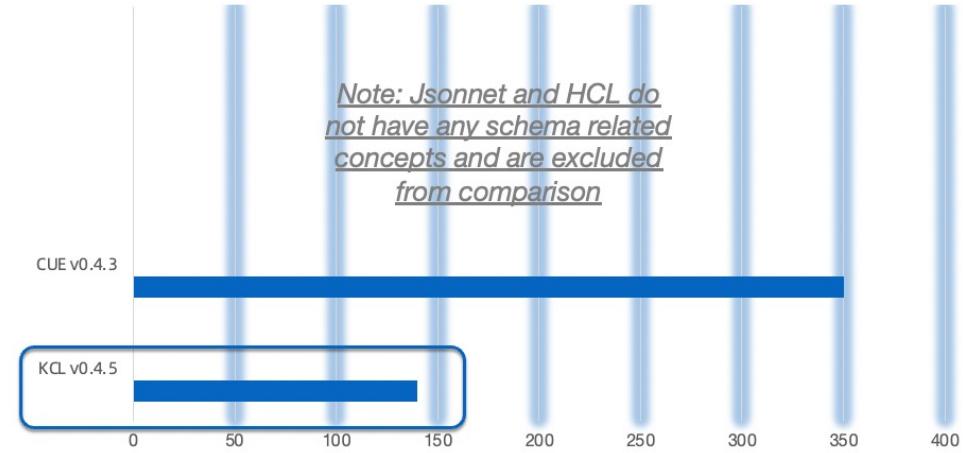
Loop and Function

```
a = lambda x: int, y: int -> int {  
    max([x, y])  
}  
temp = {"a${i}": a(1, 2) for i in range(10000)}
```



Kubernetes Configuration

```
import kubernetes.api.apps.v1  
  
deployment = v1.Deployment {}
```



*Test environment: single core macOS 10.15.7 CPU: i7-8850H
2.6GHz 32GB 2400Mhz DDR4 No NUMA, e2e run time (ms)*

性能超越社区同类型项目

为什么选择 Rust

我从前跟别人谈论Rust



我现在跟别人谈论Rust



@稀土掘金技术社区

为什么选择 Rust

Stack Machine 虚拟机实现大比拼 (循环指令进行压测, 最终 Rust 胜出)



StackMachine / src / main.py

```
Code Blame 313 lines (287 loc) · 7.82 KB Code 55% faster
```

```
70     @dataclass
71     class VM:
72         stack: List[KCLObject]
73         globals: Dict[str, KCLObject]
74         locals: Dict[str, KCLObject]
75
76         def push(self, obj: KCLObject):
77             self.stack.append(obj)
78
79         def pop(self) -> KCLObject:
80             return self.stack.pop()
81
82         def peek_nth(self, i: int) -> KCLObject:
83             return self.stack[len(self.stack)-i]
84
85         def peek(self) -> KCLObject:
86             return self.stack[len(self.stack)-1]
87
88         def run(self, bytecode: ByteCode):
89             i = 0
90             while i < len(bytecode.instructions):
91                 elem = bytecode.instructions[i]
92                 if elem.opcode == BinaryAdd:
93                     pass
94                 elif elem.opcode == StoreGlobal:
95                     # print("StoreGlobal")
96                     name = bytecode.names[elem.operand3]
97                     self.globals[name] = self.pop()
```



StackMachine / src / main.rs

```
Code Blame 382 lines (382 loc) · 11.3 KB Code 55% faster with GitHub Copilot
```

```
291     struct VM {}
292     impl VM {
293         pub fn run(self, bytecode: &ByteCode) {
294             println!("Run KCLVM Rust!");
295             let mut stack: Vec<KCLObjectRef> = vec![];
296             let mut globals: HashMap<String, KCLObjectRef> = HashMap::new();
297             let mut locals: HashMap<String, KCLObjectRef> = HashMap::new();
298             let mut i: usize = 0;
299             let mut j: usize = 0;
300             while i < bytecode.instructions.len() {
301                 let elem = &bytecode.instructions[i];
302                 match elem.opcode {
303                     Opcode::BinaryAdd => {
304                         println!("BinaryAdd");
305                     }
306                     Opcode::StoreGlobal => {
307                         // println!("StoreGlobal");
308                         let index = elem.operand3 as usize;
309                         let name = bytecode.names[index].clone();
310                         let obj = stack.pop().expect("empty stack");
311                         globals.insert(name, obj);
312                     }
313                     Opcode::StoreLocal => {
314                         // println!("StoreLocal");
315                         let index = elem.operand3 as usize;
316                         let name = bytecode.names[index].clone();
317                         let obj = stack.last().expect("last object not found");
318                         locals.insert(name, (*obj).clone());
319                     }
320                 }
321             }
322         }
323     }
```



StackMachine / src / main.go

```
Code Blame 352 lines (345 loc) · 5.99 KB Code 55% faster with GitHub Copilot
```

```
70     type ByteCode struct {
71         instructions []Instruction
72     }
73     type VM struct {
74         stack []KCLObject
75         globals map[string]KCLObject
76         locals map[string]KCLObject
77     }
78
79     func (vm *VM) push(obj KCLObject) {
80         vm.stack = append(vm.stack, obj)
81     }
82
83     func (vm *VM) pop() KCLObject {
84         last := vm.stack[len(vm.stack)-1]
85         vm.stack = vm.stack[:len(vm.stack)-1]
86         return last
87     }
88
89     func (vm *VM) peek_nth(i int) KCLObject {
90         return vm.stack[len(vm.stack)-i]
91     }
92
93     func (vm *VM) peek() KCLObject {
94         return vm.stack[len(vm.stack)-1]
95     }
96
97     func (vm *VM) run(bytecode *ByteCode) {
98         i := 0
99         for i < len(bytecode.instructions) {
100             elem := bytecode.instructions[i]
101             switch elem.opcode {
102                 case BinaryAdd:
103                     break
104                 case StoreGlobal:
```

为什么选择 Rust

- **JS/TS/Python Infra -> Rust:** 越来越多的编程语言的编译器或运行时特别是前端基础设施以及 Python 基础设施项目采用 Rust 编写或重构，此外基础设施层，数据库、搜索引擎、网络设施、云原生、UI 层和嵌入式等领域都有 Rust 的出现，至少在编程语言领域实现方面经过了可行性和稳定性验证
- **业务考量：**考虑到后续的项目 SmartIR 涉及区块链和智能合约方向，而社区中大量的区块链和智能合约项目采用 Rust 编写
- **多语言集成：**通过 Rust 获得更好的性能和稳定性，让系统更容易维护、更加健壮的同时，可以通过 FFI 暴露 C API 供多语言使用和扩展，方便生态扩展与集成
- **WASM 友好：**社区中大量 WASM 生态是由 Rust 构建，KCL 语言和编译器可以借助 Rust 编译到 WASM 并在浏览器中运行

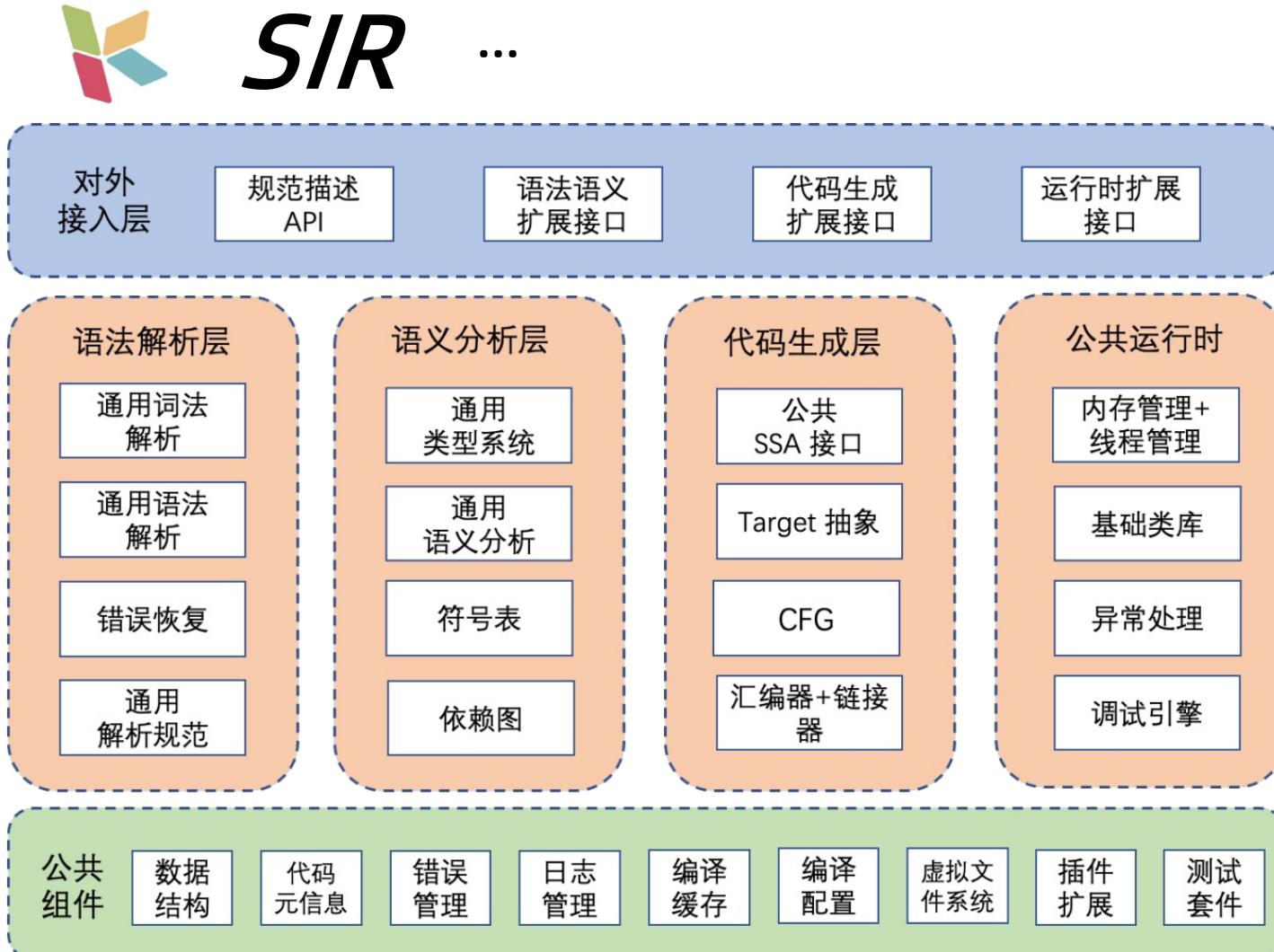
遇到了哪些困难

- **飞机上换引擎**: 边重写，边测试，边生产落地，一共 2000+ 的单元测试需要通过
- **心智转换**: Rust 某些独有特性的学习成本：函数式，所有权，生命周期，异步等，不过入门了写多了就好了
- **Unsafe**: 对待 Unsafe 代码需要小心翼翼，一不小心就跑飞
- **循环引用之殇**: KCL 语言很多数据结构都是图这种，并且语义允许变量循环引用并在 Runtime 进行垃圾回收，比如 `Rc<RefCell<KCLValue>>` 和 `Arc<Mutex<KCLValue>>` 难以设计和编写，不像别的带 GC 的语言直接写就完事了

02

演进 - 奔腾跃进 🐾

CompilerBase



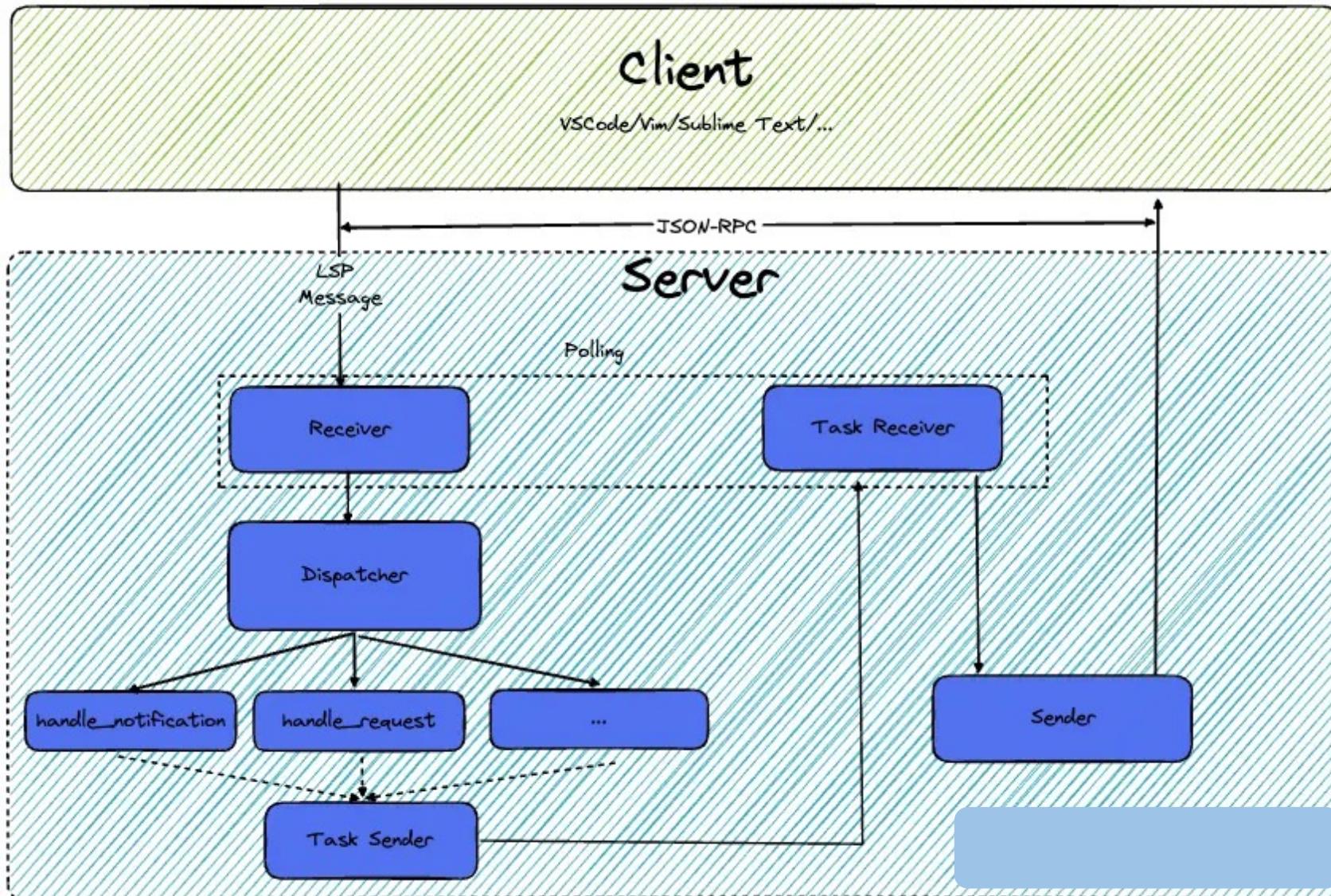
问题

- 前端的一些 Rust 基础设施如 SWC 等大量参考了 Rustc
- Rustc 项目中近乎是 Rust 的最佳实践，有些依赖太重，有些是非 Stable 的

解法

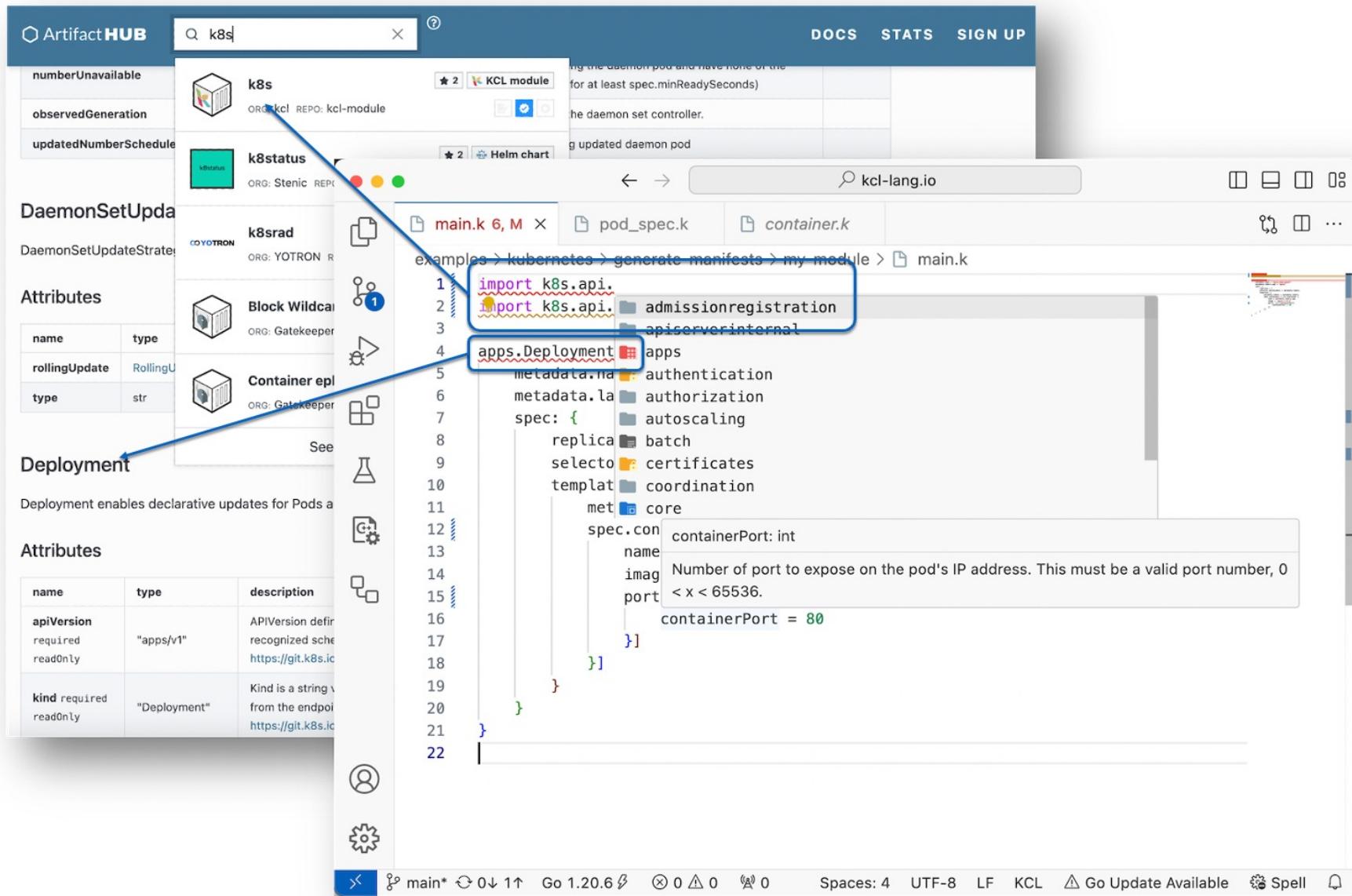
- 基于 Rustc Crates 和 Rustc 官方周边项目如错误处理、核心数据结构和算法等构建了一套编译器基础设施

Language Server



- 基于 Rust 性能依旧是几十倍几十倍的提升，用户体验有明显改善
- Language Server 借助 Rust 编译为 WASM，让 KCL 直接跑在浏览器中

IDE & Registry Based Rust LSP



多语言 SDK

二方/三方生态项目



libkcl.dylib/so/dll



KCL Rust Core

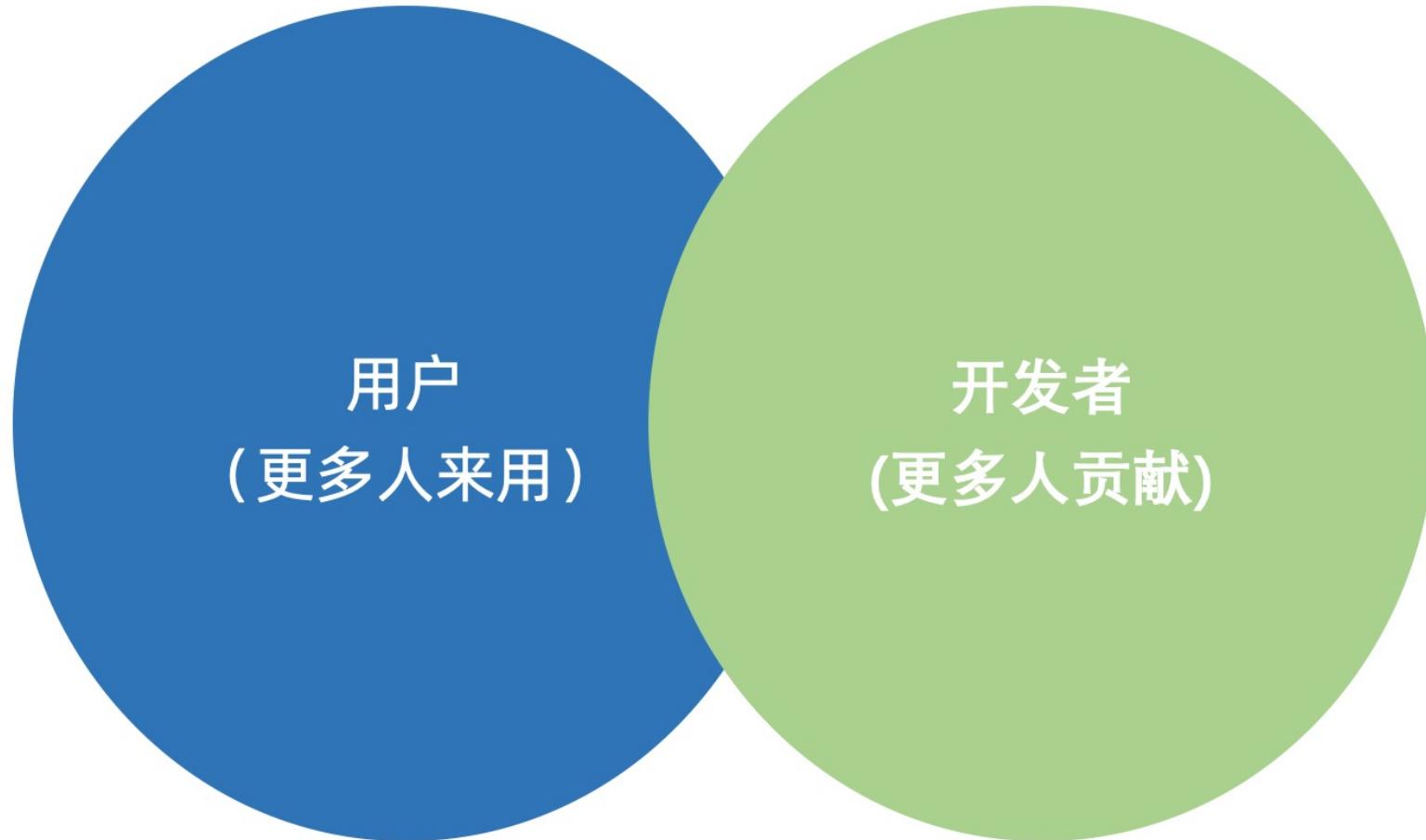
- Rust Core 编译为动态链接库，通过多语言的包管理工具将 Rust Core 打包到不同语言 SDK 中。
- 通过 Protobuf Spec 定义多语言 API, 代码自动生成, 省时省力
- Rust 多语言构建工具链生态成熟, 用 Rust 开发一个 Python 库可能比 Python 开发一个 Python 库还敏捷
- 多语言 SDK 用于不同场景的同时也赋予我们相比于其他社区产品竞争优势

03

展望 - 星辰大海



2024

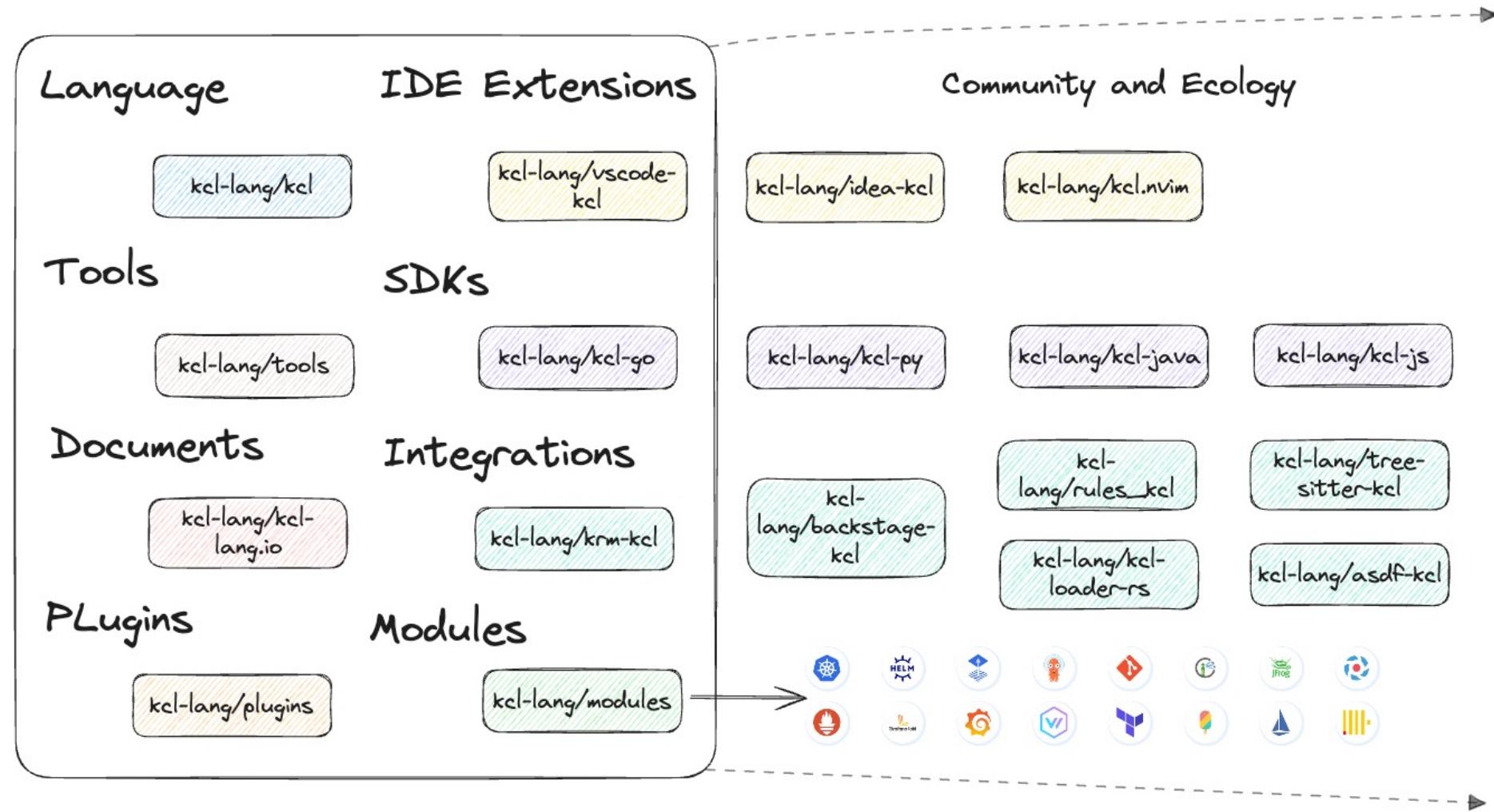


面向开源，我们坚持体验和生态优先，Rust 很好地满足了诉求，希望有越来越多的开发者参与进来

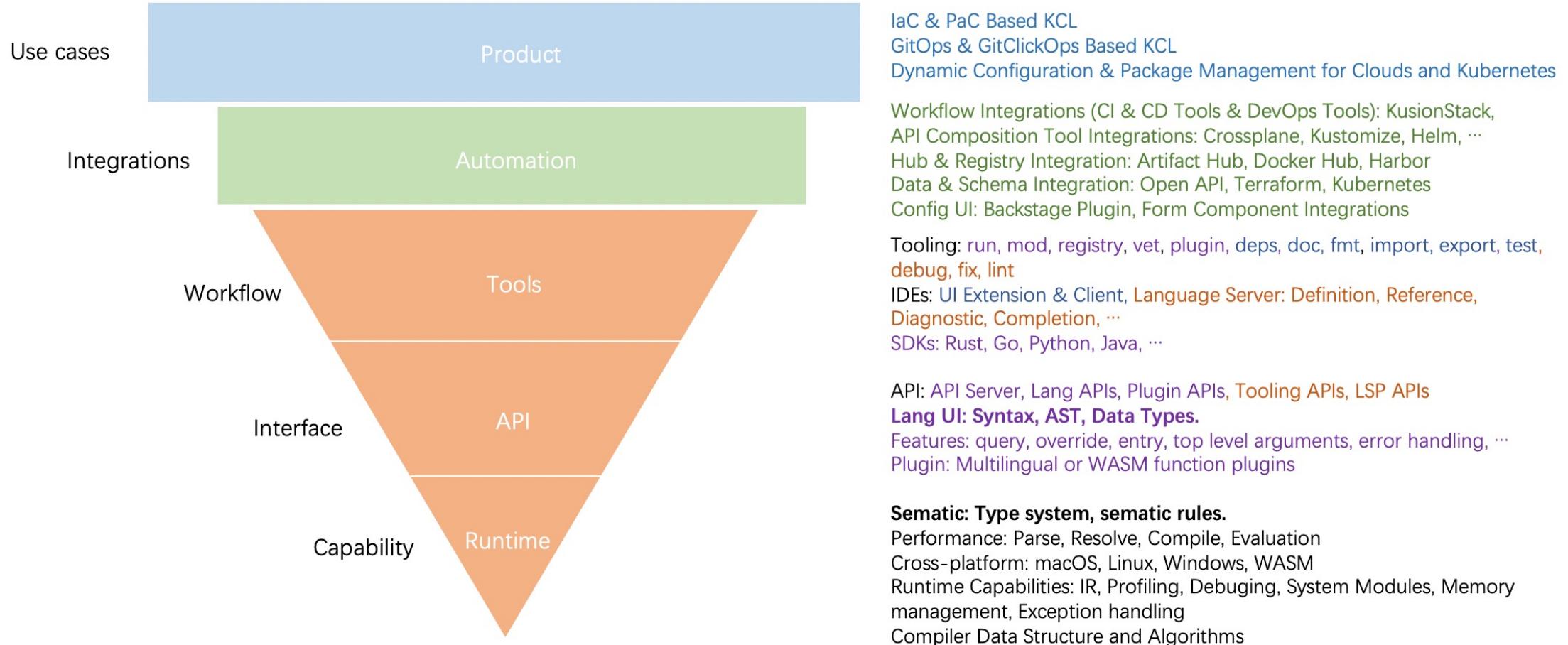
<https://github.com/kcl-lang/kcl/issues/882>

附录

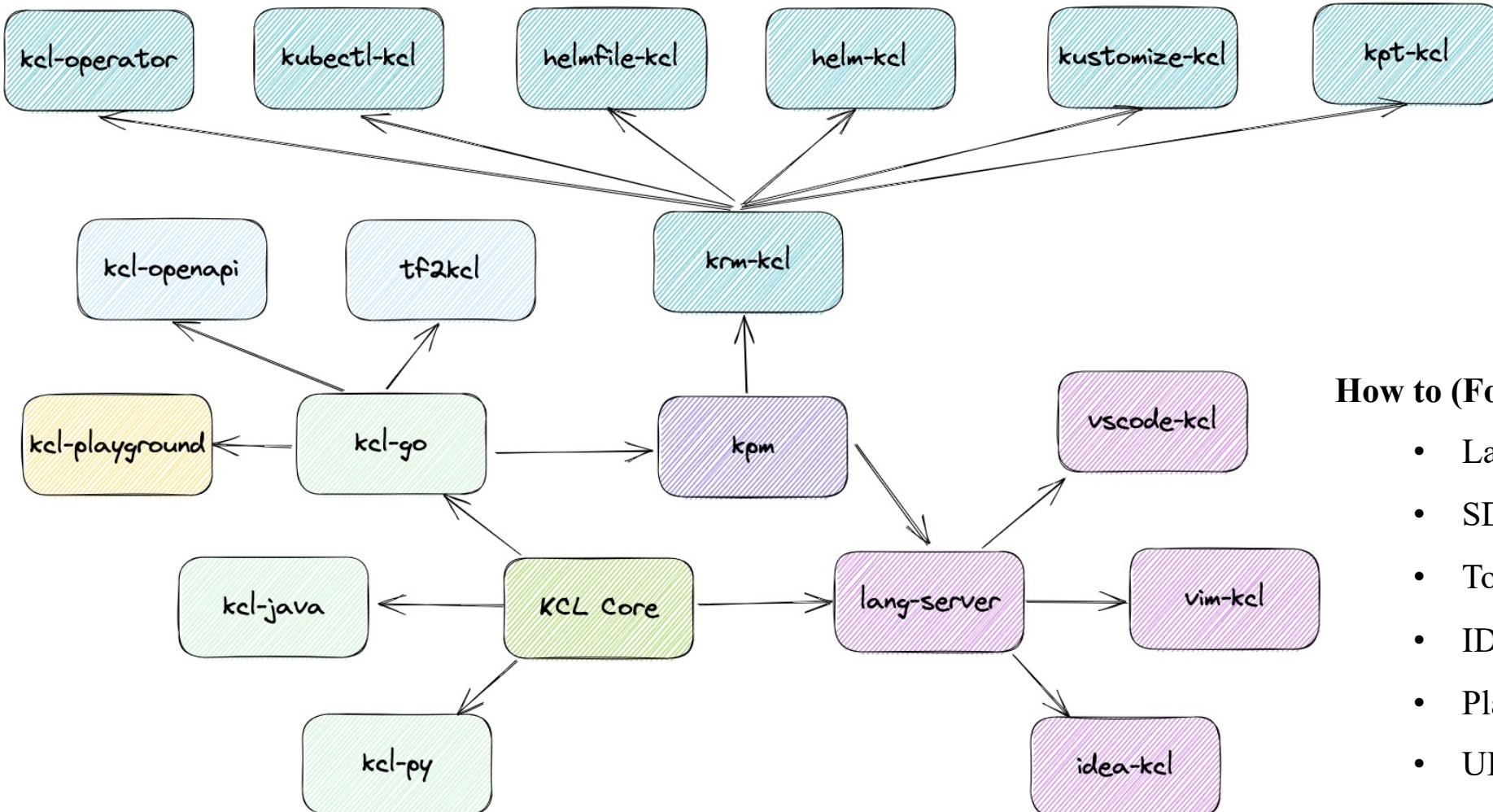
项目组织



模块组织



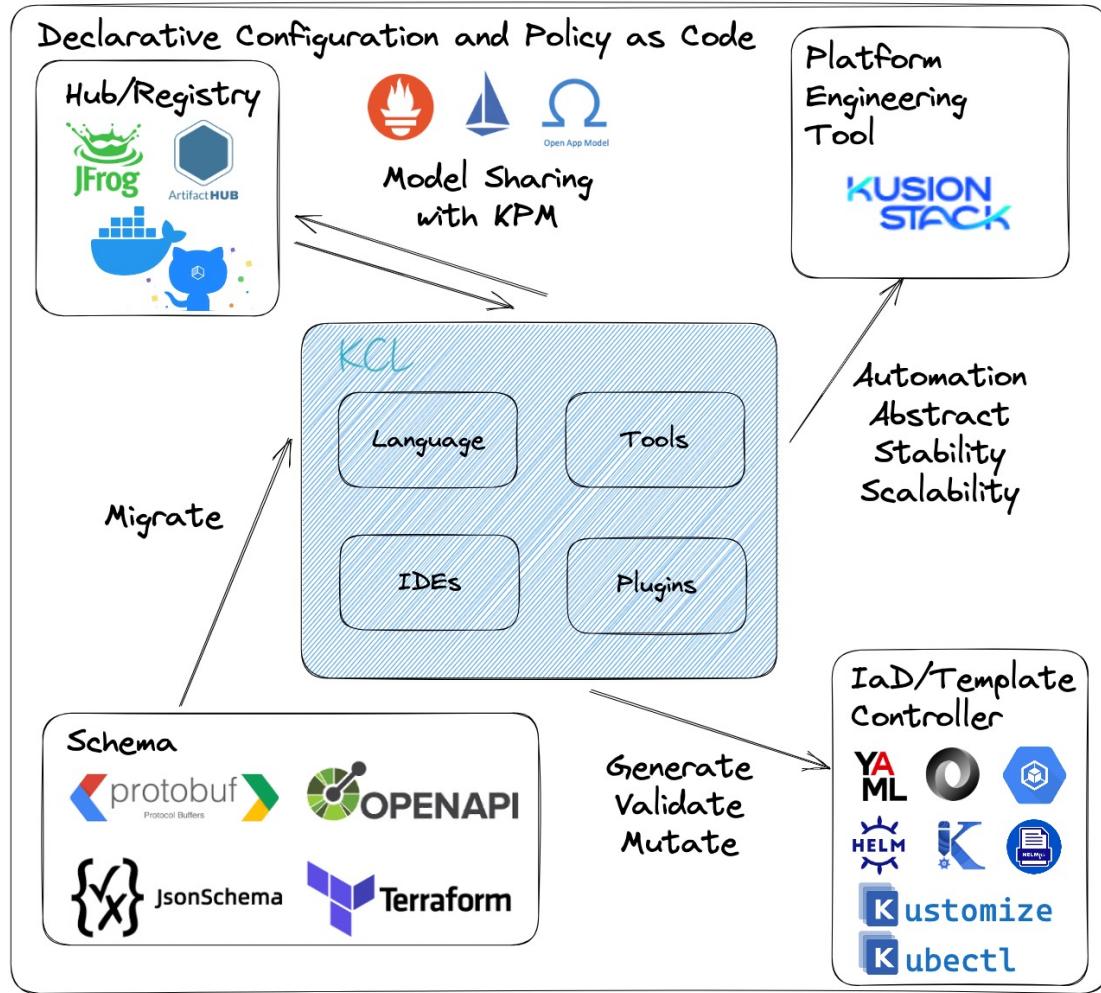
组件依赖关系



How to (For App/Platform dev & SRE)

- Lang
- SDKs
- Tools
- IDE
- Playground
- UI
- Cloud-native Tool Integrations

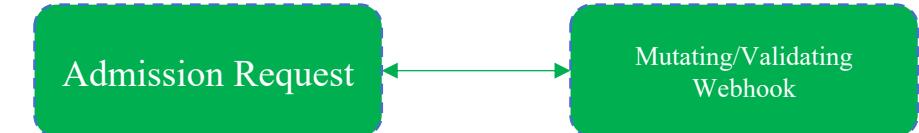
生态集成



Client



Server



Binding



Enhancement/Glue

- KRM Support: Unified KRM KCL Spec
- Migration: Data, Schema
- Sharing: Modules, Functions
- GitOps
- Lang Binding

社区产品和定位

