

Rust 编译器架构解析

王思成

Oct 20, 2026

1 为什么研究 rustc

在工业界与学术界，生产级编译器同时承担着「正确性」与「性能」双重使命。Rust 编译器 rustc 因其在编译期即完成所有权与借用检查，兼具极高的可扩展性与模块化设计，而备受关注。与 GCC、LLVM、GHC 不同，rustc 把内存安全规则提前到前端阶段，杜绝了大量运行时错误；同时，它将驱动、前端、中端、后端清晰分离，使得外部开发者能够通过独立 crate 介入任意阶段。

2 整体分层与 crate 布局

rustc 仓库分为 compiler 与 library 两大顶层目录。compiler 目录下，rustc_driver 负责命令行入口与流程编排，rustc_interface 对外暴露可编程 API，rustc_session 与 rustc_span 则共同管理诊断信息与源码位置。library 目录里，proc_macro 提供过程宏运行时，test 提供内建测试框架。

从用户输入到最终可执行文件，rustc 依次经历词法分析、语法分析、HIR 生成、类型检查、MIR 构建、LLVM IR 翻译、汇编与链接等步骤。整个管线以查询系统为核心，任何阶段的产物都可被增量缓存。

3 前端：词法与语法分析

词法分析由 rustc_lexer 手工实现，采用零拷贝设计，直接在源码切片上迭代。遇到 Unicode 标识符、raw 字符串或 raw 字节字符串时，lexer 会先判断前缀，再按需消费定界符与转义序列，避免中间缓冲区。

语法解析位于 rustc_parse，使用 LALRPOP 生成的 LALR(1) 分析表。错误恢复采用「恐慌模式」：当遇到无法规约的 token 时，解析器跳过至最近的同步 token，再补一个占位节点，使后续分析继续。生成的 AST 以递归枚举形式存在，顶层为 ast::Crate，其下依次是 Mod、Item、Expr 等节点；每个节点携带 NodeId 与 Span，前者用于跨阶段引用，后者用于精确定位源码区间。

4 语义分析与类型系统

名称解析由 rustc_resolve 完成。它首先建立模块树，再解析 use 别名与宏作用域。对于待解析的路径，解析器会依次尝试当前模块、父模块与外部 crate，最终把路径映射为 DefId。

早期类型检查由 rustc_hir_analysis 承担。它首先把 AST 降级为 HIR，剥离语法糖并插入自动引用/解引用。借用检查基于两阶段模型：第一阶段在 MIR 上收集「出借」与「使用」约束，第二阶段交给约束求解器判断是

否满足生命周期不相交规则。Polonius 作为实验性实现，采用全新的基于事实的约束模型，可处理更复杂的控制流。

特质求解目前部分集成 Chalk 引擎。Chalk 把 where 子句视为逻辑谓词，通过 SLG 归结算法寻找解；关联类型规范化则在求解过程中延迟进行，避免过早决定具体类型。

5 MIR：安全与优化的分水岭

MIR 的设计目标是显式控制流与去语法糖，使数据流分析变得直接。从 HIR/THIR 生成 MIR 的过程由 `rustc_mir_build` 负责：它首先建立作用域树，记录变量声明与 Drop 顺序；随后把表达式翻译成基本块序列，每个基本块内只包含简单语句。

借用检查在 MIR 层面再次执行。当前默认使用两阶段检查器：先在 CFG 上收集「借用起点—使用点」对，再用约束图求解；Polonius 则把借用关系编码为事实库，通过 Datalog 规则推导是否违例。MIR 优化位于 `rustc_mir_transform`，包括死代码消除、常量传播与去虚化等 pass，每一个 pass 都以函数为单位独立运行。

6 后端：LLVM 与多目标抽象

代码生成管线首先由 `rustc_codegen_ssa` 把 MIR 映射到目标无关的中间表示，再由 `rustc_codegen_llvm` 翻译为 LLVM IR。`rustc_target` 负责目标三元组的解析，包含 `spec`、`target_feature` 与链接参数；LTO、PGO、BOLT 等优化均在此阶段通过 LLVM 的扩展接口接入。

除 LLVM 外，rustc 还支持 Cranelift 与 GCC 后端，分别位于 `cg_clif` 与 `cg_gcc` crate。运行时组件如 panic 与格式化字符串，由 `rustc_borrowck` 提供的 `lang item` 实现；它们在 MIR 阶段即被特殊标记，最终在后端生成对应的运行时代码。

7 诊断、增量编译与工具链生态

诊断系统由 `rustc_errors` 驱动，支持多行错误、修复建议与 JSON 输出。每个错误都携带错误代码，文档测试会自动校验代码示例中的诊断信息是否与预期一致。

增量编译依赖查询系统与依赖图。每个查询以 `DefId` 或类型为键，计算结果附带指纹；当源码或依赖变化时，仅受影响的查询被重算。`rust-analyzer` 通过 `crates.io` 上的 `ra-ap-rustc` 包复用 rustc 的名称解析与类型检查；`clippy` 与 `miri` 则直接调用 `rustc_interface` 提供的回调接口，实现自定义 lint 与解释执行。

rustc 的设计为语言设计者提供了「渐进式复杂性」与「早期严格检查」的范例。未来，GAT、`return position impl Trait in traits` 以及 Polonius 的正式上线，将进一步扩展类型系统与借用检查的表达能力。任何对编译器感兴趣的开发者，都可以通过 `good first issues` 或 `mentorship` 群组参与 rustc 开发，共同完善这一现代系统级语言的基础设施。