## Rust 拥有出色的文档、友好的编译器和清晰的错误提示信息， 还集成了一流的工具——包管理器和构建工具， 智能地自动补全和类型检验的多编辑器支持， 以及自动格式化代码等等。

## [说明](https://rustmagazine.github.io/rust_magazine_2021/chapter_1/rustc_part1.html" \l "%E8%AF%B4%E6%98%8E)

《图解 Rust 编译器与语言设计》系列文章特点：

1. 重在图解。图解的目的，是为了帮助开发者从整体结构、语义层面来掌握 Rust 编译器与语言设计。
2. 边实践边总结，不一定会每月都有，但争取吧。
3. 希望是众人合力编写，我只是抛砖引玉。硬骨头，一起啃。

## [引子](https://rustmagazine.github.io/rust_magazine_2021/chapter_1/rustc_part1.html" \l "%E5%BC%95%E5%AD%90)

想必读者朋友们都已经看到了 《Rust 日报》里的消息：微软、亚马逊、Facebook等巨头，都在组建自己的 Rust 编译器团队，都在战略性布局针对 Rust 语言。并且 Rust 基金会也已经进入了最后都流程，由此可以猜想，这些巨头很可能已经加入了基金会。

我在 RustChinaConf 2020 年大会分享《Rust 这五年》中盘点了 Rust 这五年多都发展，虽然 Rust 势头很好，但大部分贡献其实都是国外社区带来的，国内社区则是处于学习和观望的状态，等待着所谓的杀手级应用出现来引领 Rust 的“走红”。为什么国内社区不能为 Rust 多做点实质性的贡献呢？

因此，2020 新年到来的时候，我立下一个五年的 Flag ： 五年内要为 Rust 语言发 1000 个 PR。

然后社区里的朋友就帮我做了一个计算：五年 1000 个，那么每年 200 个，那么一天就得 0.5 个。也有朋友说，Rust 的 PR 每次 Review 周期都很长，就算你能一年提 200 个 PR，官方也不可能给你合并那么多。

这样的计算，确实很有道理。这个目标，确实很难完成。但其实这个 Flag 我并没有打算个人完成，而是想推动社区对 Rust 感兴趣对朋友一起完成。如果五年内，我能推动 1000 个人参与，那么每个人只提交一个 PR，那么这个 1000 个 PR 的 Flag 就轻松完成了。

所以，为了完成这个 Flag ，我把未来五年划分成三个阶段：

1. 第一阶段：2021 年。该阶段的目标是「上道」。
2. 第二阶段：2022 ～ 2023 年。该阶段的目标是「进阶」。
3. 第三阶段：2024 ~ 2025 年。该阶段目标是「达标」。

也就是说，今年是想要「上道」的一年。那么要达成这个目标，我做了以下计划：

1. 组织社区力量来翻译官方的《Rust 编译器开发指南》。
2. 组织 Rust 编译器小组，开始为 Rust 语言做点贡献，并且将在此过程中自己的学习和经验沉淀为《图解 Rust 编译器与语言设计》系列文章。

通过这两份文档，希望可以帮助和影响到更多的人，来为 Rust 语言做贡献。

我知道，编译器作为程序员的三大浪漫之一，水很深。你也可能会说，人家搞编译器的都是 PL 出生，一般人哪有那种本事。诚然如你所想，编译器很难。但幸亏，难不等于不可能。不会，我们可以学。况且，也不是让你从零开始去实现一个 Rust 编译器。

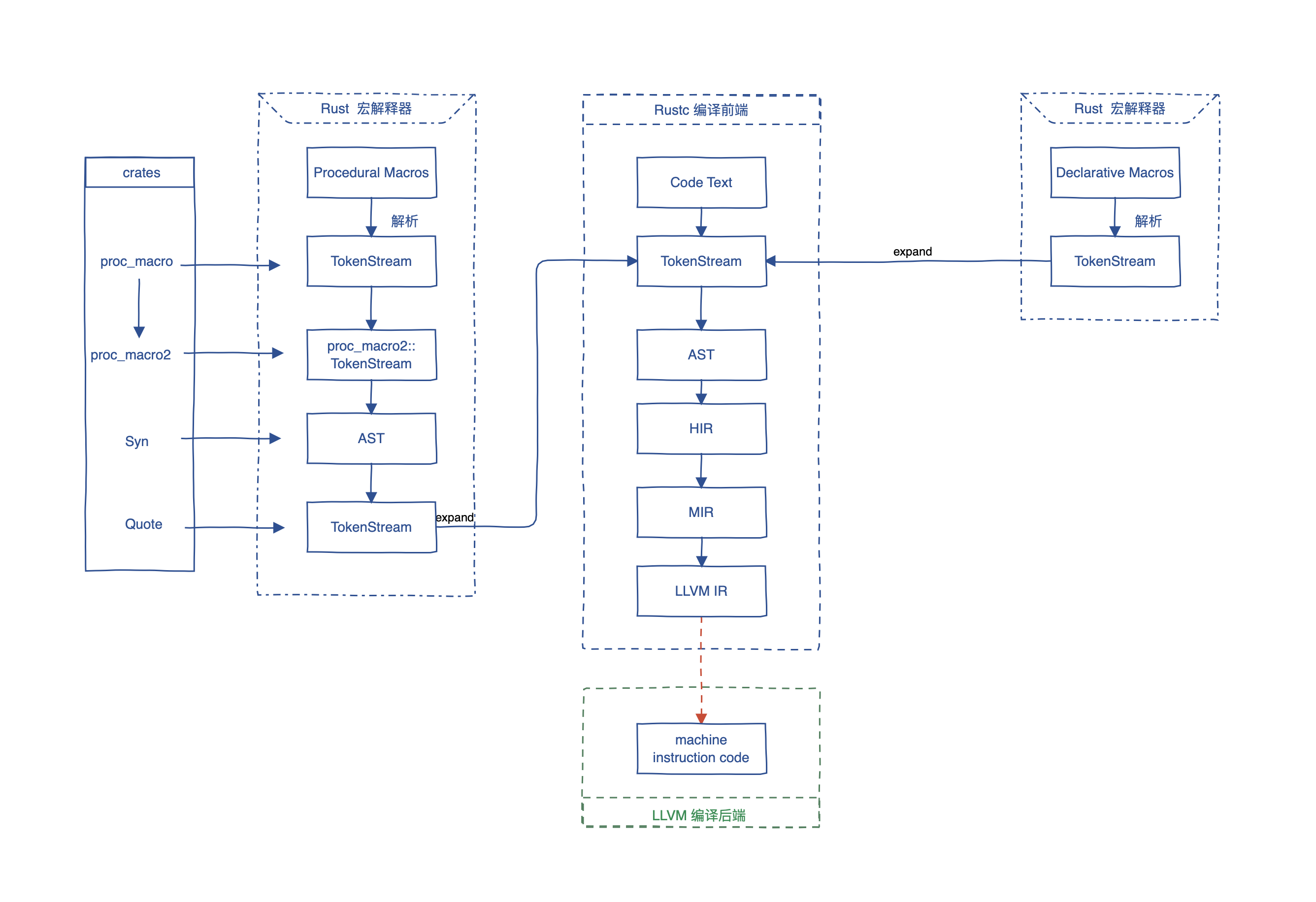
为 Rust 语言做贡献，并不是 KPI 驱动，而是兴趣驱动。可能你看完了编译原理龙书虎书鲸书三大经典，也可能你实现过自己的一门语言。但其收获可能永远也比不上实际参与到 Rust 这样一个现代化语言项目中来。

所以，《图解 Rust 编译器与语言设计》系列文章，不仅仅会记录我自己学习 Rust 编译器的沉淀，还会记录你的沉淀，如果你愿意投稿的话。在这浮躁的世界，给自己一片净土，找回技术初心。

## [图解 Rust 编译过程](https://rustmagazine.github.io/rust_magazine_2021/chapter_1/rustc_part1.html" \l "%E5%9B%BE%E8%A7%A3-rust-%E7%BC%96%E8%AF%91%E8%BF%87%E7%A8%8B)

对于学习，我通常习惯先从整体和外围下手，去了解一个东西的全貌和结构之后，再逐步深入细节。否则的话，很容易迷失到细节中。

所以，必须先来了解 Rust 编译过程。如下图：



上图中间部分为 Rust 代码的整体编译过程，左右两边分别为过程宏和声明宏的解释过程。

Rust 语言是基于 LLVM 后端实现的编程语言。在编译器层面来说，Rust编译器仅仅是一个编译器前端，它负责从文本代码一步步编译到LLVM中间码(LLVM IR)，然后再交给LLVM来最终编译生成机器码，所以LLVM就是编译后端。

### [Rust 语言编译整体流程](https://rustmagazine.github.io/rust_magazine_2021/chapter_1/rustc_part1.html" \l "rust-%E8%AF%AD%E8%A8%80%E7%BC%96%E8%AF%91%E6%95%B4%E4%BD%93%E6%B5%81%E7%A8%8B)

1. Rust 文本代码首先要经过「词法分析」阶段。

将文本语法中的元素，识别为对 Rust 编译器有意义的「词条」，即token。

经过词法分析之后，再通过语法分析将词条流转成「抽象语法树（AST）」。

在得到 AST 之后，Rust 编译器会对其进行「语义分析」。

一般来说，语义分析是为了检查源程序是否符合语言的定义。在 Rust 中，语义分析阶段将会持续在两个中间码层级中进行。

1. 语义分析 HIR 阶段。

HIR 是抽象语法树（AST）对编译器更友好的表示形式，很多 Rust 语法糖在这一阶段，已经被脱糖（desugared）处理。比如 for 循环在这个阶段会被转为loop，if let 被转为match，等等。HIR 相对于 AST 更有利于编译器的分析工作，它主要被用于 「类型检查（type check）、推断（type inference）」。

1. 语义分析 MIR 阶段。

MIR 是 Rust 代码的中级中间代表，基于 HIR 进一步简化构建。MIR 是在RFC 1211中引入的。

MIR 主要用于借用检查。早期在没有 MIR 的时候，借用检查是在 HIR 阶段来做的，所以主要问题就是生命周期检查的粒度太粗，只能根据词法作用域来进行判断，导致很多正常代码因为粗粒度的借用检查而无法通过编译。Rust 2018 edition 中引入的 非词法作用域生命周期（NLL）就是为来解决这个问题，让借用检查更加精细。NLL 就是因为 MIR 的引入，将借用检查下放到 MIR 而出现的一个术语，这个术语随着 Rust 的发展终将消失。

MIR 这一层其实担负的工作很多，除了借用检查，还有代码优化、增量编译、Unsafe 代码中 UB 检查、生成LLVM IR等等。关于 MIR 还需要了解它的三个关键特性：

* 它是基于控制流图（编译原理：Control Flow Graph）的。
* 它没有嵌套表达式。
* MIR 中的所有类型都是完全明确的，不存在隐性表达。人类也可读，所以在 Rust 学习过程中，可以通过查看 MIR 来了解 Rust 代码的一些行为。

图中没有画出来的，还有一个从 HIR 到 MIR 的一个过渡中间代码表示 THIR（Typed HIR） 。THIR 是对 HIR 的进一步降级简化，用于更方便地构建 MIR 。在源码层级中，它属于 MIR 的一部分。

生成 LLVM IR 阶段。LLVM IR是LLVM中间语言。LLVM会对LLVM IR进行优化，再生成为机器码。

后端为什么要用 LLVM ？不仅仅是 Rust 使用 LLVM，还有很多其他语言也使用它，比如 Swift 等。 LLVM 的优点：

* LLVM后端支持的平台很多，我们不需要担心CPU、操作系统的问题（运行库除外）。
* LLVM后端的优化水平较高，我们只需要将代码编译成LLVM IR，就可以由LLVM后端作相应的优化。
* LLVM IR本身比较贴近汇编语言，同时也提供了许多ABI层面的定制化功能。

Rust 核心团队也会帮忙维护 LLVM，发现了 Bug 也会提交补丁。虽然LLVM有这么多优点，但它也有一些缺点，比如编译比较慢。所以，Rust 团队在去年引入了新的后端 Cranelift ，用于加速 Debug 模式的编译。Rust 编译器内部组件 rustc\_codegen\_ssa 会生成后端无关的中间表示，然后由 Cranelift 来处理。从2021年1月开始，通过rustc\_codegen\_ssa 又为所有后端提供了一个抽象接口以实现，以允许其他代码源后端（例如 Cranelift），这意味着，Rust 语言将来可以接入多个编译后端（如果有的话）。

以上是 Rust 整体编译流程。但 Rust 语言还包含来强大的元编程：「宏（Macro）」，宏代码是如何在编译期展开的呢？请继续往下看。

### [Rust 宏展开](https://rustmagazine.github.io/rust_magazine_2021/chapter_1/rustc_part1.html" \l "rust-%E5%AE%8F%E5%B1%95%E5%BC%80)

Rust 本质上存在两类宏：**声明宏（Declarative Macros）** 与 **过程宏（Procedural Macros）** 。很多人可能搞不清楚它们的差异，也许看完这部分内容就懂了。

**声明宏**

回头再看看上面的图右侧部分。我们知道，Rust 在最初解析文本代码都时候会将代码进行词法分析生成词条流（TokenStream）。在这个过程中，如果遇到了宏代码（不管是声明宏还是过程宏），则会使用专门的「宏解释器（Macro Parser）」 来解析宏代码，将宏代码展开为 TokenStream，然后再合并到普通文本代码生成的 TokenSteam 中。

你可能会有疑问，其他语言的宏都是直接操作 AST ，为什么 Rust 的宏在 Token 层面来处理呢？

这是因为 Rust 语言还在高速迭代期，内部 AST 变动非常频繁，所以无法直接暴露 AST API 供开发者使用。而词法分析相对而言很稳定，所以目前 Rust 宏机制都是基于词条流来完成的。

那么声明宏，就是完全基于词条流（TokenStream)。声明宏的展开过程，其实就是根据指定的匹配规则（类似于正则表达式），将匹配的 Token 替换为指定的 Token 从而达到代码生成的目的。因为仅仅是 Token 的替换（这种替换依然比 C 语言里的那种宏强大），所以你无法在这个过程中进行各种类型计算。

**过程宏**

声明宏非常方便，但因为它只能做到替换，所以还是非常有局限的。所以后来 Rust 引入了过程宏。过程宏允许你在宏展开过程中进行任意计算。但我们不是说，Rust 没有暴露 AST API 吗？为什么过程宏可以做到这么强大？

其实，过程宏也是基于 TokenSteam API的，只不过由第三方库作者 dtolnay 设计了一套语言外的 AST ，经过这一层 AST 的操作，就实现了想要的结果。

没有什么问题不是可以通过加一层解决的，如果解决不了那就加两层。

dtolnay 在社区内被誉为最佳 API 设计天才。他创造了不少库，比如 Serde，是 Rust 生态中被应用最多的一个库。

话说回来。过程宏的工作机制就如上面图中左侧展示的那样。主要是利用三个库，我称之为 「过程宏三件套」：

1. proc\_macro2。该库是对 proc\_macro 的封装，是由 Rust 官方提供的。
2. syn。该库是 dtolnay 实现的，基于 proc\_macro2 中暴露的 TokenStream API 来生成 AST 。该库提供来方便的 AST 操作接口。
3. quote。该库配合 syn，将 AST 转回 TokenSteam，回归到普通文本代码生成的 TokenSteam 中。

过程宏的整个过程，就像是水的生态循环。 蒸汽从大海（TokenSteam）中来，然后通过大雨(Syn)，降到地上(Quote)，形成涓涓细流(proc\_macro2::TokenStream)最终汇入大海（TokenSteam）。

理解过程宏的展开原理，将有助于你学习过程宏。

## [小结](https://rustmagazine.github.io/rust_magazine_2021/chapter_1/rustc_part1.html" \l "%E5%B0%8F%E7%BB%93)

本篇文章主要介绍了 Rust 代码的编译过程，以及 Rust 宏代码的展开机制，学习这些内容，将有助于你深入理解 Rust 的概念。不知道这篇内容是否激发起你对 Rust 编译器对兴趣呢？编译器是一个深坑，让我们慢慢挖掘它。