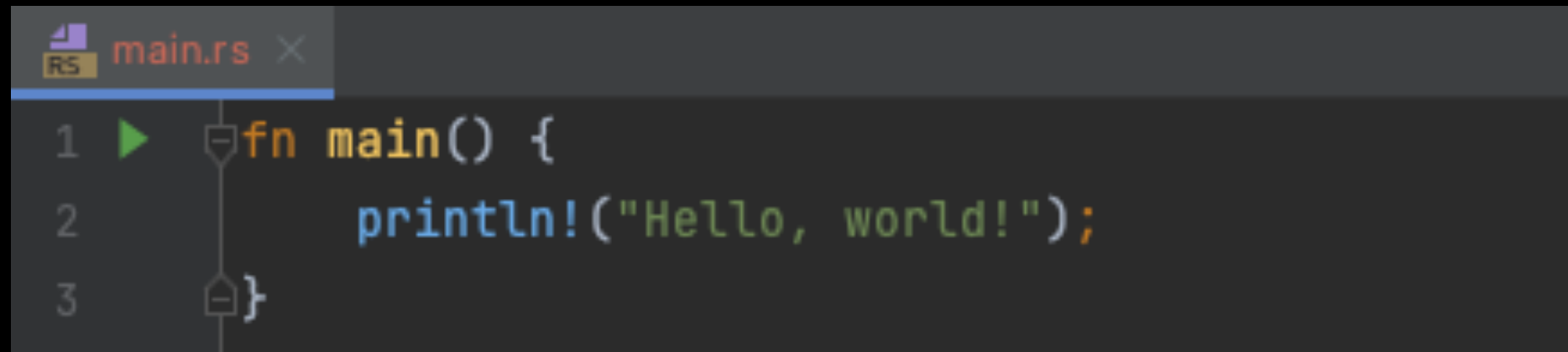


通过实战理解Rust宏

探讨一个长期以来对我来说相当陌生的话题：宏

苏林

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏



```
main.rs x
1  ▶  fn main() {
2      println!("Hello, world!");
3  }
```

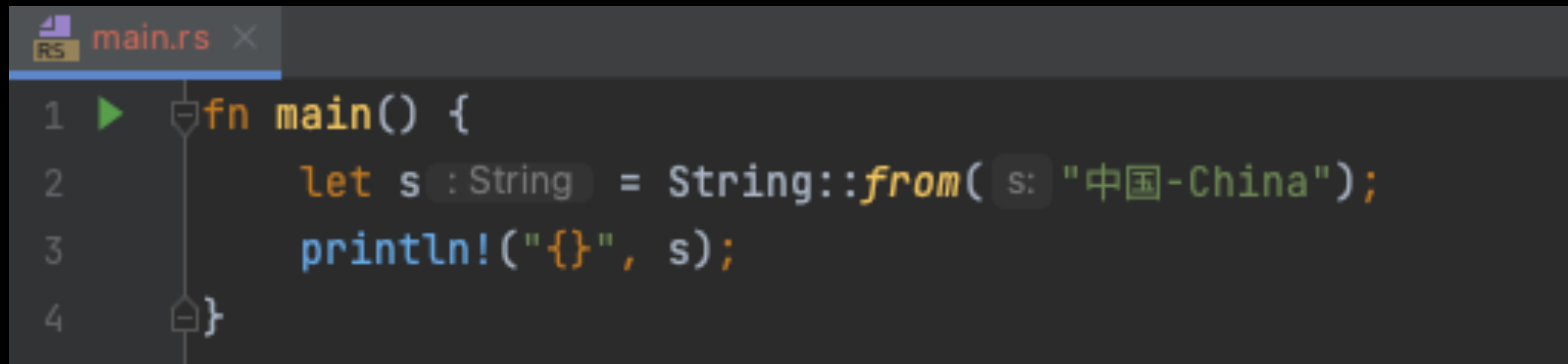
The image shows a code editor window with a tab labeled 'main.rs'. The code is written in Rust and consists of three lines: a function signature 'fn main() {' on line 1, an indented call to 'println!("Hello, world!");' on line 2, and a closing brace '}' on line 3. A green play button icon is next to line 1, and a vertical line with a small square at the bottom indicates the current cursor position at the end of line 3.

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

```
main.rs x
1  ▶  fn main() {
2      println!("Hello, world!");
3  }
```

```
83  ///
84  /// # Examples
85  ///
86  /// ```
87  /// println!(); // prints just a newline
88  /// println!("hello there!");
89  /// println!("format {} arguments", "some");
90  /// ```
91  #[macro_export]
92  #[stable(feature = "rust1", since = "1.0.0")]
93  #[allow_internal_unstable(print_internals, format_args_nl)]
94  macro_rules! println {
95      () => ($crate::print!("\n"));
96      ($($arg:tt)*) => ({
97          $crate::io::_print($crate::format_args_nl!($($arg)*));
98      })
99 }
```

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏



```
main.rs x
1  ▶  fn main() {
2      let s : String = String::from(s: "中国-China");
3      println!("{}", s);
4      }
```

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

```
main.rs x
1  ▶  fn main() {
2      let s : String = String::from(s: "中国-China");
3      println!("{}", s);
4  }
```

```
285  /// [`str`]: prim@str
286  /// [`&str`]: prim@str
287  /// [`Deref`]: core::ops::Deref
288  /// [`as_str()`]: String::as_str
289  #[derive(PartialOrd, Eq, Ord)]
290  #[cfg_attr(not(test), rustc_diagnostic_item = "string_type")]
291  #[stable(feature = "rust1", since = "1.0.0")]
292  ↓  pub struct String {
293      vec: Vec<u8>,
294  }
```

Example

A basic TCP echo server with Tokio.

Make sure you activated the full features of the tokio crate on Cargo.toml:

```
[dependencies]
tokio = { version = "1.10.0", features = ["full"] }
```

Then, on your main.rs:

```
use tokio::net::TcpListener;
use tokio::io::{AsyncReadExt, AsyncWriteExt};

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            let mut buf = [0; 1024];

            // In a loop, read data from the socket and write the data back.
            loop {
                let n = match socket.read(&mut buf).await {
                    // socket closed
```

自我介绍

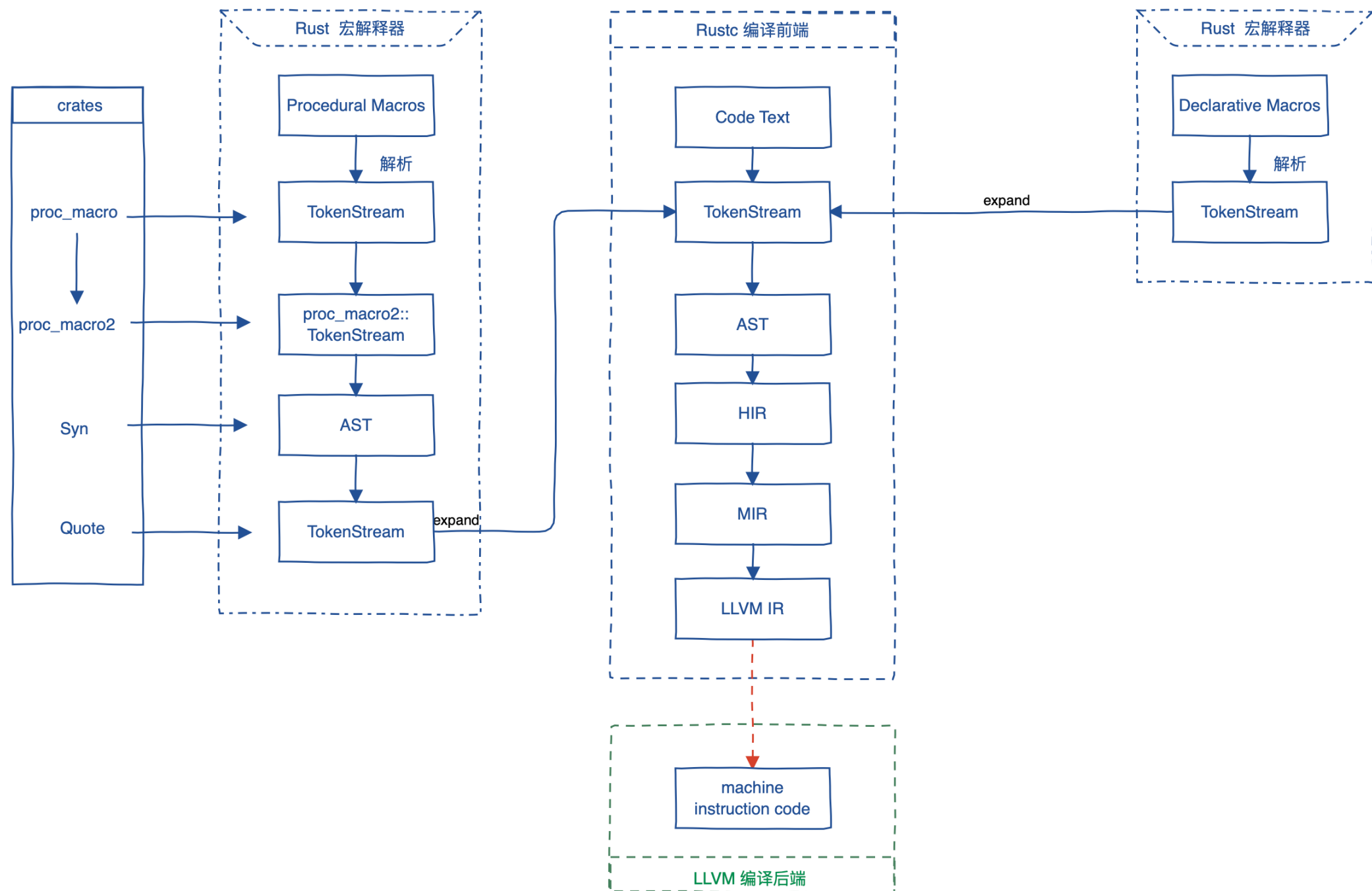
- 前折800互联网研发团队负责人, 10余年一线研发经验
- 目前是多点Dmall技术Leader
- 具有多年的软件开发经验, 精通Ruby、Java、Rust等开发语言
- 同时也参与过Rust中文社区日报维护工作.

分享内容

- 一起重温Rust整个编译过程
- 声明宏
- 实战过程宏
- 过程宏的学习资料

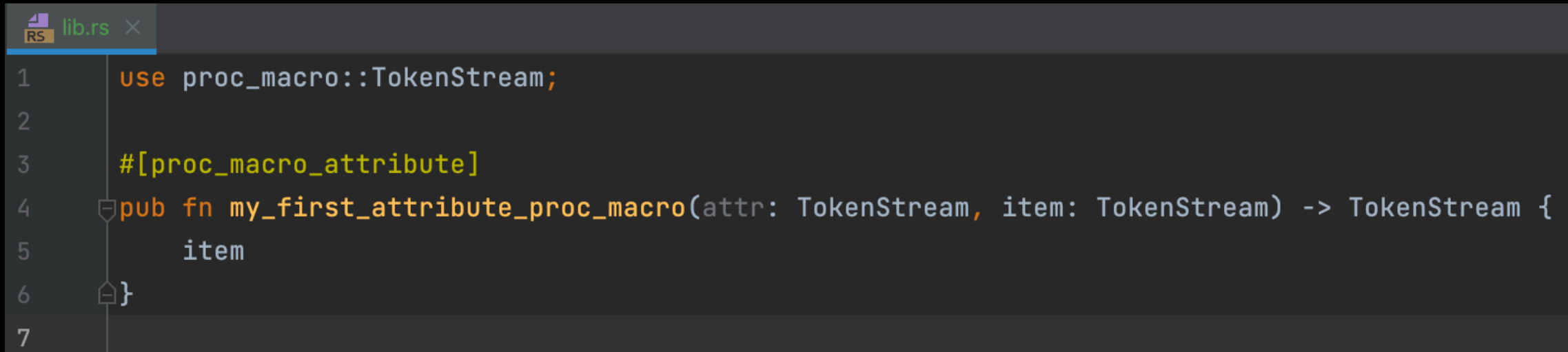
通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

Rust编译过程



通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

结合着刚才讲的 Rust 过程宏解释器, 能理解以下代码的入参和出参吗?



```
1 use proc_macro::TokenStream;
2
3 #[proc_macro_attribute]
4 pub fn my_first_attribute_proc_macro(attr: TokenStream, item: TokenStream) -> TokenStream {
5     item
6 }
7
```

结合着刚才讲的 Rust 过程宏解释器, 能理解以下代码的入参和出参吗?

```
lib.rs x
1 use proc_macro::TokenStream;
2
3 #[proc_macro_attribute]
4 pub fn my_first_attribute_proc_macro(attr: TokenStream, item: TokenStream) -> TokenStream {
5     item
6 }
7
```

```
170 /// Note that `start_paused` requires the `test-util` feature to be enabled.
171 ///
172 /// ### NOTE:
173 ///
174 /// If you rename the Tokio crate in your dependencies this macro will not work.
175 /// If you must rename the current version of Tokio because you're also using an
176 /// older version of Tokio, you _must_ make the current version of Tokio
177 /// available as `tokio` in the module where this macro is expanded.
178 #[proc_macro_attribute]
179 #[cfg(not(test))] // Work around for rust-lang/rust#62127
180 pub fn main(args: TokenStream, item: TokenStream) -> TokenStream {
181     entry::main(args, item, true)
182 }
```

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

结合着刚才讲的 Rust 过程宏解释器, 能理解以下代码的入参和出参吗?

```
8      #[proc_macro_derive(FirstDeriveProcMacro)]
9      pub fn my_first_derive_proc_macro(input: TokenStream) -> TokenStream {
10         TokenStream::new()
11     }
```

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

声明宏

<https://kaisery.github.io/trpl-zh-cn/ch19-06-macros.html>

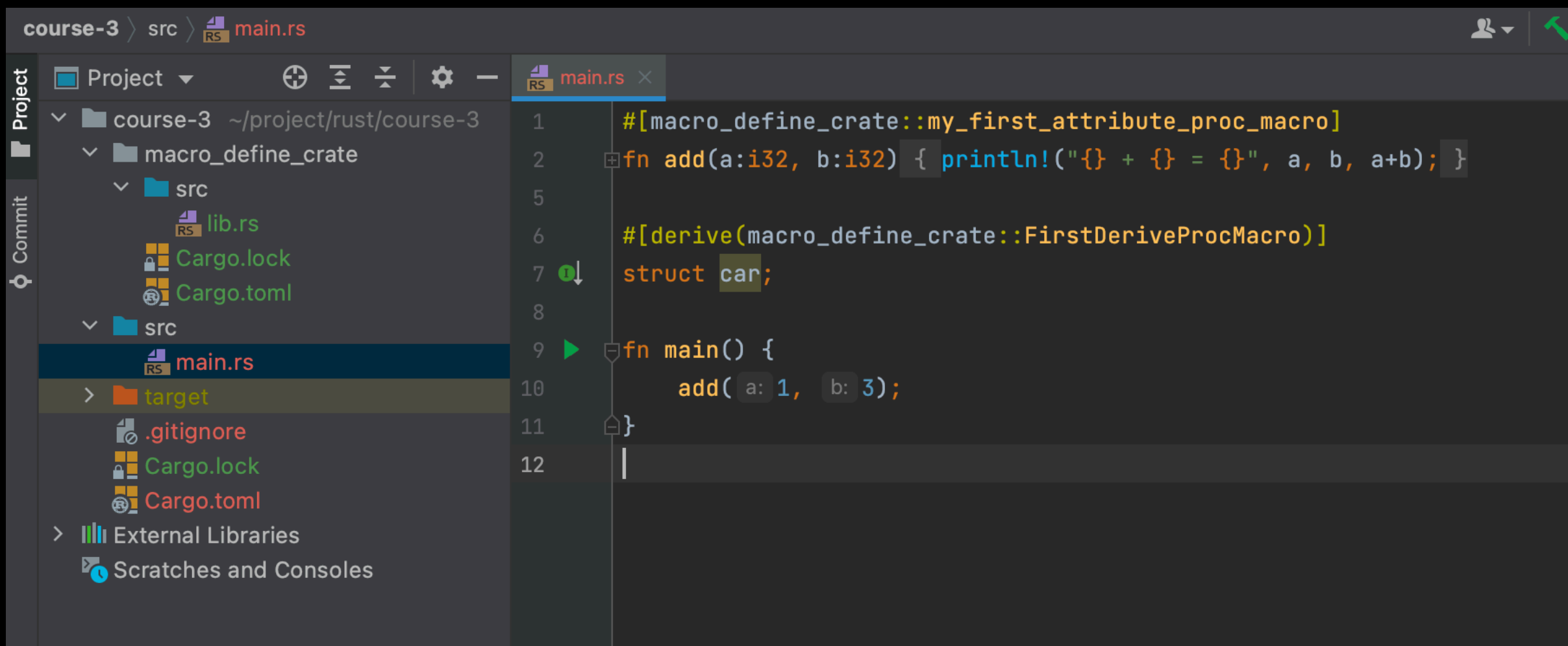
实战过程宏

如果想用Rust去开发大型项目，特别是框架级别的项目，那么Rust的过程宏(proc-macro)机制肯定是一个必须掌握的技能。

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

实战过程宏

如果想用Rust去开发大型项目，特别是框架级别的项目，那么Rust的过程宏(proc-macro)机制肯定是一个必须掌握的技能。



The screenshot shows an IDE window for a Rust project named 'course-3'. The left sidebar displays the project structure, including a 'macro_define_crate' directory with a 'src' subdirectory containing 'lib.rs', 'Cargo.lock', and 'Cargo.toml'. The main editor area shows the 'main.rs' file with the following code:

```
1  #[macro_define_crate::my_first_attribute_proc_macro]
2  fn add(a:i32, b:i32) { println!("{}", a, b, a+b); }
5
6  #[derive(macro_define_crate::FirstDeriveProcMacro)]
7  struct car;
8
9  fn main() {
10     add(a: 1, b: 3);
11 }
12
```

实战过程宏

探讨一个问题, 这也是我刚学习过程宏的时候困扰过我的问题.

为什么过程宏必须定义在一个独立的crate中, 难道不能在一个crate中既定义过程宏, 又使用过程宏?

实战过程宏

探讨一个问题, 这也是我刚学习过程宏的时候困扰过我的问题.

为什么过程宏必须定义在一个独立的crate中, 难道不能在一个crate中既定义过程宏, 又使用过程宏?

原理: 考虑过程宏是在编译一个crate之前, 对crate的代码进行加工的一段程序, 这段程序也是需要编译后执行的。如果定义过程宏和使用过程宏的代码写在一个crate中, 那就陷入了死锁:















-> 要编译的代码首先需要运行过程宏来展开, 否则代码是不完整的, 没法编译crate.

-> 不能编译crate, crate中的过程宏代码就没法执行, 就不能展开被过程宏装饰的代码.

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

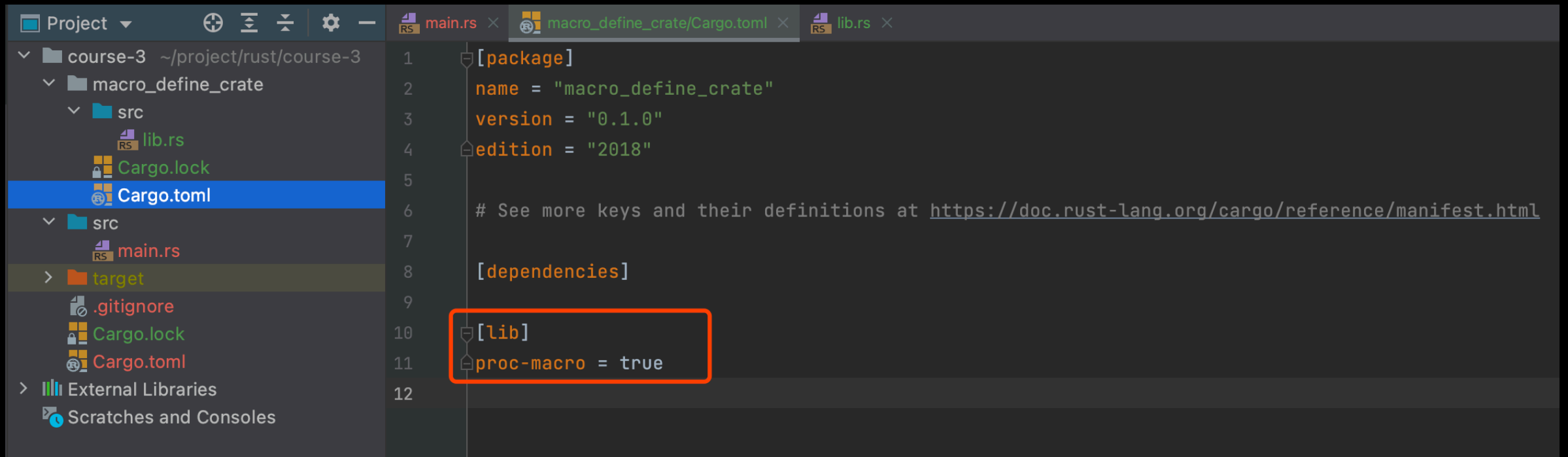
实战过程宏

正是由于这个原理, 我们在学习开源项目时, 都能很容易找到定义宏的源码.

| | | | |
|---|---|---|-----------------|
|  | Darksonn ci: add readme check to CI (#4039) | ✓ f478647 2 days ago | 🕒 2,610 commits |
|  | .github | ci: add readme check to CI (#4039) | 2 days ago |
|  | benches | bench: update spawn benchmarks (#3927) | last month |
|  | bin | chore: script updating versions in links to docs.rs (#1249) | 2 years ago |
|  | examples | net: add read/try_read etc methods to NamedPipeServer (#3899) | 2 months ago |
|  | stress-test | deps: update rand to 0.8, loom to 0.4 (#3307) | 8 months ago |
|  | tests-build | chore: fix CI on master (#4008) | 16 days ago |
|  | tests-integration | runtime: fix memory leak/growth when creating many runtimes (#3564) | 5 months ago |
|  | tokio-macros | chore: explicitly relaxed clippy lint for runtime entry macro (#4030) | 4 days ago |
|  | tokio-stream | chore: fix doc failure in CI on master (#4020) | 12 days ago |
|  | tokio-test | fs: document performance considerations (#3920) | last month |
|  | tokio-util | chore: fix clippy warnings (#4017) | 12 days ago |
|  | tokio | chore: prepare Tokio v1.10.0 (#4038) | 3 days ago |
|  | .cirrus.yml | util: remove path deps (#3413) | 7 months ago |

实战过程宏

在macro_define_crate/Cargo.toml中添加[lib]节点并在下面增加proc-macro = true



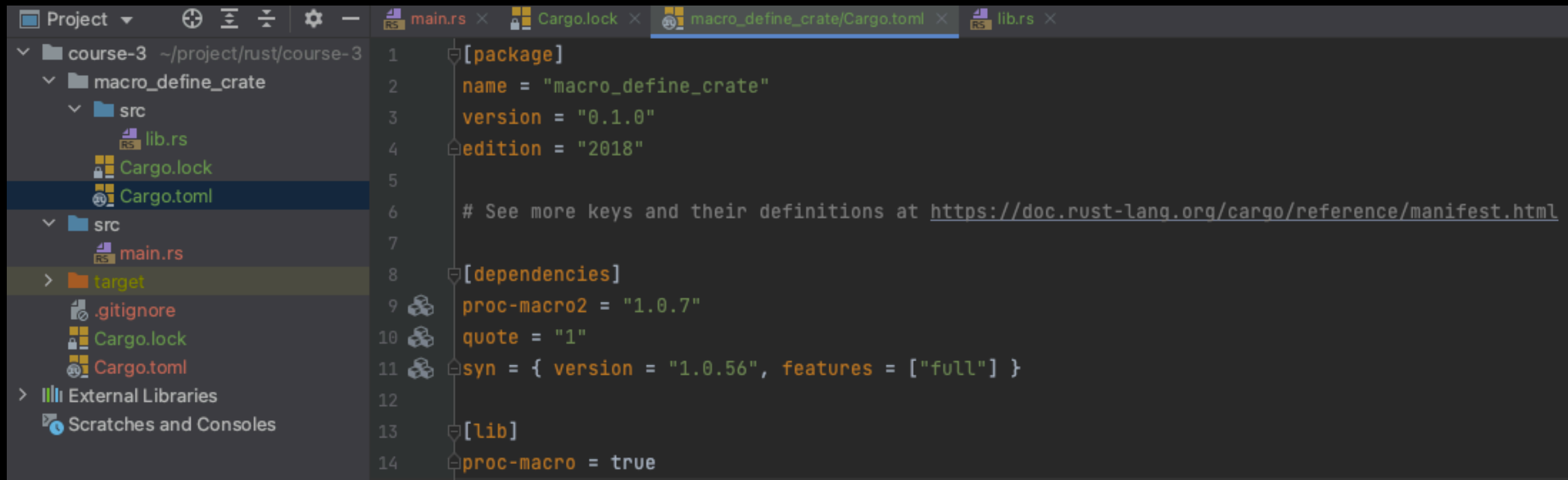
```
1 [package]
2   name = "macro_define_crate"
3   version = "0.1.0"
4   edition = "2018"
5
6   # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
7
8   [dependencies]
9
10  [lib]
11    proc-macro = true
12
```

表示这个crate是一个proc-macro，增加这个配置以后，这个crate的特性就会发生一些变化，例如，这个crate将只能对外导出内部定义的过程宏，而不能导出内部定义的其他内容。

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

实战过程宏

在dependencies节点下，添加如下三个依赖包，这也是开发rust过程宏必备的三个依赖包。



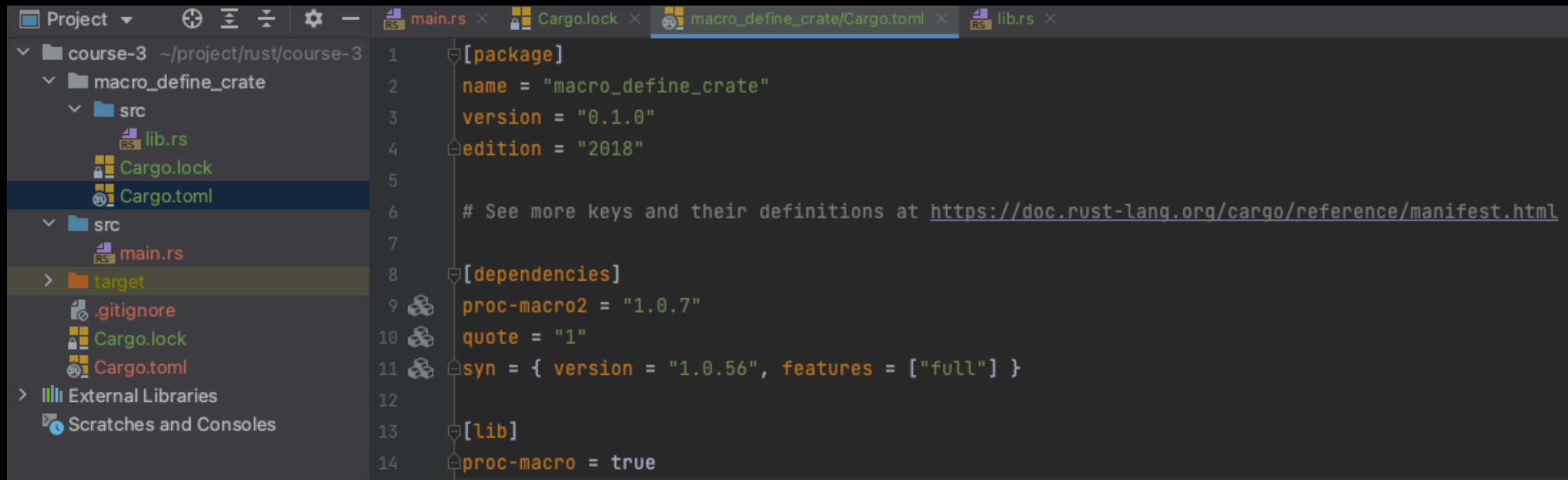
The screenshot shows an IDE with a project named 'course-3' at the path '~/project/rust/course-3'. The file explorer on the left shows the project structure, including a 'macro_define_crate' directory with 'src' and 'target' subdirectories. The 'Cargo.toml' file is selected in the 'src' directory. The main editor displays the contents of 'macro_define_crate/Cargo.toml'.

```
1  [package]
2    name = "macro_define_crate"
3    version = "0.1.0"
4    edition = "2018"
5
6    # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
7
8  [dependencies]
9    proc-macro2 = "1.0.7"
10   quote = "1"
11   syn = { version = "1.0.56", features = ["full"] }
12
13  [lib]
14    proc-macro = true
```

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

实战过程宏

在dependencies节点下，添加如下三个依赖包，这也是开发rust过程宏必备的三个依赖包。

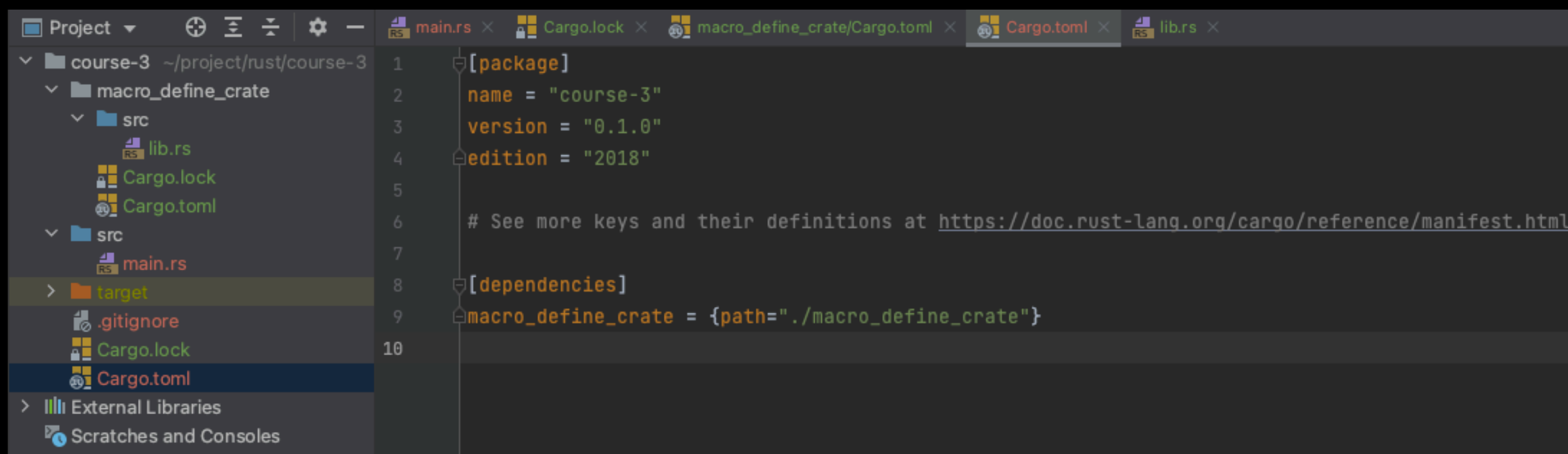


The screenshot shows an IDE with a project named 'course-3' at the path '~/project/rust/course-3'. The file explorer on the left shows the project structure, including a 'macro_define_crate' directory with 'src' and 'target' subdirectories. The 'Cargo.toml' file is selected in the 'src' directory. The main editor displays the contents of 'macro_define_crate/Cargo.toml'.

```
1  [package]
2    name = "macro_define_crate"
3    version = "0.1.0"
4    edition = "2018"
5
6    # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
7
8  [dependencies]
9    proc-macro2 = "1.0.7"
10   quote = "1"
11   syn = { version = "1.0.56", features = ["full"] }
12
13  [lib]
14    proc-macro = true
```

实战过程宏

打开外层crate的course-3/Cargo.toml文件，添加上面的内层crate作为依赖。我们已经说过了，这两个crate本质上是独立的，这里仅仅是通过依赖关系把它们给关联起来了。只要下面这两行配置文件中的路径写的对，这两个包在磁盘上存在哪个目录下都行。



The screenshot shows an IDE with a project explorer on the left and a code editor on the right. The project explorer shows a directory structure for 'course-3' with subdirectories 'macro_define_crate' and 'src'. The 'macro_define_crate' directory contains 'lib.rs', 'Cargo.lock', and 'Cargo.toml'. The 'src' directory contains 'main.rs'. The 'target' directory is also visible. The code editor shows the 'Cargo.toml' file with the following content:

```
1  [package]
2    name = "course-3"
3    version = "0.1.0"
4    edition = "2018"
5
6    # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
7
8  [dependencies]
9    macro_define_crate = {path="./macro_define_crate"}
10
```

实战过程宏

`[proc_macro_attribute]`是在告诉编译器我们在定义一个“属性式”的过程宏

`#[proc_macro]`和`#[proc_macro_derive]`,分别用于定义“函数式”和“派生式”两种类型的过程宏

Rust过程宏的本质就是一个编译环节的过滤器，或者说是一个中间件，它接收一段用户编写的源代码，做一通酷炫的操作，然后返回给编译器一段经过修改的代码。

rust的过程宏代码目前没有比较好的调试方法，`print`大法应该是最好用的。

实战过程宏

TokenStream以树形结构的数据组织，表达了用户源代码中各个语言元素的类型以及相互之间的关系

每个语言元素都有一个span属性，记录了这个元素在用户源代码中的位置。

不同类型的节点，有各自独有的属性，其他我们在这个例子中没有涉及到的节点类型，大家可以去阅读文档

Ident类型表示的是一个标识符，用的非常频繁的一个类型。变量名、函数名等等，都是标识符。

TokenStream里面的信息，是没有语义信息的，比如在上面的例子中，路径表达式中的双冒号::被拆分为两个独立的冒号对待，TokenStream并没有把它们识别为路径表达式，同样，它也不区分这个冒号是出现在一个引用路径中，还是用来表示数据类型。

针对attr属性而言，其中不包括宏自己的名称的标识符，它包含的仅仅是传递给这个过程宏的参数信息

实战过程宏

现在就很容易理解了：所谓的Rust过程宏，就是我们可以自己修改上面的item变量中的值，从而等价于加工原始输入代码,最后将加工后的代码返回给编译器即可。

实战过程宏

回顾 `syn`、`quote`、`proc_macro2` 这些包

`parse_macro_input!(attr as AttributeArgs)`输出的是`Vec<NestedMeta>`类型的语法树节点

`parse_macro_input!(item as Item)`输出的是`Item`类型的语法树节点

通过`quote::quote!`宏，可以将语法树节点及其子节点重新转换为`TokenStream`

`quote::quote!`宏返回的结果实际是`proc_macro2::TokenStream`类型的数据。

通过`into()`进行一下转换，变成`proc_macro::TokenStream`类型的数据

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

实战过程宏

为什么需要 `syn`、`quote`、`proc_macro2` 这些包来进行转换？

实战过程宏

为什么需要 `syn`、`quote`、`proc_macro2` 这些包来进行转换？

这是有一定的历史背景的

由于Rust语言还处在年幼阶段，编译器内部结构变化也会比较大，所以rust编译器给我们提供的接口是没有语义的，近似于一堆字符串片段的`proc_macro::TokenStream`类型的数据。这个`proc_macro`包是由rust官方提供的。

`quote`、`syn`、`proc_macro2`这些包，都是由社区或者第三方提供的，这些包基于官方的`proc_macro::TokenStream`做了很多额外的功能，目的是为了更方便我们更方便地去处理、编辑、生成`proc_macro::TokenStream`

从这一点来体会，rust提供了标准（`proc_macro::TokenStream`），通过这些标准，可以在上层发展各种生态（`quote`、`syn`、`proc_macro2`）来扩展rust。

通过实战理解Rust宏 | 探讨一个长期以来对我来说相当陌生的话题：宏

实战过程宏

再来手写一个 https://github.com/denoland/flaky_test

学习资料

proc-macro-workshop项目

戴维·托尔奈（David Tolnay，也就是syn和quote这两个库的作者）的教学项目proc-macro-workshop.

proc-macro-workshop是一个包含5个过程宏的“解题游戏”，每一个过程宏都是有实际应用价值的案例，通过一系列由简到繁的测试用例，指导你去完成每一个过程宏的开发，而我们要做的，就是编写代码，像闯关游戏一样依次通过每一个测试用例，在这个过程中，我们会学到不同类型的过程宏的开发方法.

<https://github.com/dtolnay/proc-macro-workshop>

QA环节

加群一起交流Rust & Datafuse

