

2024.04.08-2024.04.14-work-log

工作进展

本阶段完成的任务有：编写marco_main过程宏，使得用户使用rtsmart-std库时能够使用Rust标准的main函数，属性宏将用户编写的主函数转换为以 C ABI 调用约定为基础的 `main` 函数，作为程序的入口点。

本阶段创建了一个新项目marco_main，这是一个过程宏项目。使用这个宏标注main函数，这样就能在no_std模式下正常使用main函数。

由于在没有Rust标准库的支持下，也就是no_std模式下，是无法使用main函数的。于是就另辟蹊径，通过过程宏的方式对用户编写的代码进行重构，通过使用C ABI调用格式的main函数，就可以正常使用main函数。

资料收集

Rust参考手册：<https://rustwiki.org/zh-CN/reference/procedural-macros.html>

Rust宏小册：<https://zjp-cn.github.io/tlborm/>

学习编写过程宏的方法

marco_main项目代码

Cargo.toml

```
[package]
name = "marco_main"
version = "0.1.0"
edition = "2021"

[lib]
proc_macro = true

[dependencies]
quote = "1.0"
proc-macro2 = "1.0"
darling = "0.13.0"

[dependencies.syn]
version = "1.0"
features = ["extra-traits", "full"]
```

一个普通的过程宏项目的Cargo.toml文件中，在lib里标注proc_macro = true

同时需要用到quote、proc-macro2、syn等库。darling库用于自动将属性宏上的属性值转换到结构体对象上。

proc-macro2

`proc-macro2` 是对 `proc_macro` 的包装，根据其文档，它用于两个特定目的：

- 将类似与过程宏的功能带到其他上下文中，如 `build.rs` 和 `main.rs`
- 让过程宏可进行单元测试

由于 `proc_macro` 只能在 `proc-macro` 类型的库中使用，所以无法直接使用 `proc_macro` 库。

始终记住，`proc-macro2` 模仿 `proc_macro` 的 api，对后者进行包装，让后者的功能在非 `proc-macro` 类型的库中也能使用。

因此，建议基于 `proc-macro2` 来开发过程宏代码的库，而不是基于 `proc_macro` 构建，因为这将使这些库可以进行单元测试，这也是以下列出的库传入和返回 `proc-macro2::TokenStream` 的原因。

当需要 `proc_macro::TokenStream` 时，只需对 `proc-macro2::TokenStream` 进行 `.into()` 操作即可获得 `proc_macro` 的版本，反之亦然。

使用 `proc-macro2` 的过程宏通常会以别名的形式导入，比如使用 `use proc-macro2::TokenStream as TokenStream2` 来导入 `proc-macro2::TokenStream`。

quote

`quote` 主要公开了一个声明宏：`quote!`。

这个小小的宏让你轻松创建标记流，使用方法是将实际的源代码写出为 Rust 语法。

同时该宏还允许你将标记直接插入到编写的语法中：

1. 使用 `#local` 语法进行插值，其中 `local` 指的是当前作用域中的一个 local。
2. 使用 `##local` 来对实现了 `ToTokens` 的类型的迭代器进行插值，其工作原理类似于声明宏的反复，因为它们允许在反复中使用分隔符和额外的标记。

```
let name = /* 某个标识符 */;
let exprs = /* 某个对表达式标记流的迭代器 */;
let expanded = quote! {
    impl SomeTrait for #name { // #name 将插入上述的局部名称
        fn some_function(&self) -> usize {
            #( #exprs )+* // 通过迭代生成表达式
        }
    }
};
```

在准备输出时，`quote!` 是一个非常有用的工具，它避免了通过逐个插入标记来创建标记流。

syn

`syn` 是一个解析库，用于将 Rust 标记流解析为 Rust 源代码的语法树。

它是一个功能非常强大的库，使得解析过程宏输入变得非常容易，而 `proc_macro` 本身不公开任何类型的解析功能，只公开标记。

由于这个库可能是一个严重的编译依赖项，它大量使用 `feature` 控制来允许用户根据需要将其功能剪裁得尽可能小。

那么，它能提供什么呢？很多东西。

首先，当启用 `full` feature 时，它具有对所有标准 Rust 语法节点的定义和从而能够完全解析 Rust 语法。

在启用 `derive` feature（默认开启）之后，它还提供一个 `DeriveInput` 类型，该类型封装了传递给 `derive` 宏输入所有信息。

在启用 `parsing` 和 `proc-macro` feature（默认开启）之后，`DeriveInput` 可以直接与 `parse_macro_input!` 配合使用，以将标记流解析为所需的类型。

如果 Rust 语法不能解决你的问题，或者说你希望解析自定义的非 Rust 语法，那么这个库还提供了一个通用的[解析 API][`parse`]，主要是以 `Parse` trait 的形式（这需要 `parsing` feature，默认启用）。

除此之外，该库公开的类型保留了位置信息和 `Span`，这让过程宏发出详细的错误消息，指向关注点的宏输入。

由于这又是一个过程宏的库，它利用了 `proc-macro2` 的类型，因此可能需要转换成 `proc_macro` 的对应类型。

marco_main_use宏代码逻辑

首先在函数头上使用`#[proc_macro_attribute]`进行标注

同时接受两个参数

```
pub fn marco_main_use(args: TokenStream, input: TokenStream) -> TokenStream
```

- 第一个参数是属性名称后面的带分隔符的标记树，不包括它周围的分隔符。如果只有属性名称（其后不带标记树，比如 `#[attr]`），则这个参数的值为空。
- 第二个参数是添加了该过程宏属性的条目，但不包括该过程宏所定义的属性。因为这是一个 `active` 属性，在传递给过程宏之前，该属性将从条目中剥离出来。

接下来首先是使用`parse_macro_input!`宏通过Rust的代码标记流获取到源代码语法树，以及解析出过程宏上提供的`appname`和`desc`等属性

```
let f = parse_macro_input!(input as syn::ItemFn);
let raw_arg = parse_macro_input!(args as syn::AttributeArgs);
let parg = Args::from_list(&raw_arg).map_err(|e| e.write_errors());
let arg = match parg {
    Ok(x) => x,
    Err(e) => {
        return e.into();
    }
};
```

然后根据源代码语法树和args判断当前代码是否符合转换的条件

```
if arg.appname.is_none() {
    return parse::Error::new(
        Span::call_site(),
        "`#[marco_main_use]` macro must have attribute `appname`",
    )
    .to_compile_error()
    .into();
}
```

```
// check the function signature
let valid_signature = f.sig.constness.is_none()
    && f.sig.unsafety.is_none()
    && f.sig.asyncness.is_none()
    && f.vis == Visibility::Inherited
    && f.sig.abi.is_none()
    && f.sig.generics.params.is_empty()
    && f.sig.generics.where_clause.is_none()
    && f.sig.variadic.is_none()
    && match f.sig.output {
        ReturnType::Default => true,
        _ => false,
    };
if !valid_signature {
    return parse::Error::new(
        f.span(),
        "`#[entry]` function must have signature `fn(arg: vec::IntoIter<&
[u8]>)`",
    )
    .to_compile_error()
    .into();
};
```

符合条件之后，就可以通过`quote!()`宏构造新的标记流，按照之前的设计，将用户编写的主函数转换为以 C ABI 调用约定为基础的 `main` 函数，作为程序的入口点。通过以下方式直接转换然后返回

```
let content = f.block.into_token_stream();
let core = quote!(
    #[no_mangle]
    pub extern "C" fn main(_argc: isize, _argv: *const *const u8) -> usize {
        #content
        0
    }
);

quote!(
    #core
).into()
```

这种类型的主函数代码即可被成功 `aarch64-rtsmart-muslabi-gcc` 编译成可执行文件，既不违反 rust 在 `no_std` 模式下不能定义主函数的规定，又让编译器找到程序入口位置。

总结

本周的工作进展主要是编写了 `marco_main` 这一过程宏，将用户编写的主函数转换为以 C ABI 调用约定为基础的 `main` 函数，作为程序的入口点，解决 `no_std` 状态下无法使用主函数的窘境。基于这个过程宏，用户就可以正常使用主函数，配合 `rtsmart_std` 库使用，达到 Rust 标准库的使用体验。

下周或下下周我们计划会编译一个简单的 rust 程序，观察代码能否成功编译。并在 `qemu` 上测试一下标准输出库和 `marco_main` 过程宏是否可用。