So you want to build rust procedural macro with with wasm

by mav3ri3k

Contents

§1. I am clueless, lets start!

Where do you start ?

Well I don't see that the compiler allows it. Hmm.. I will do it myself then. :0

§1.1. How do do you create procedural macros ?

Lets start where all things rust start: The Rust Book → 19. Advances Features → 19.5 Macros

It teaches us a nice way to create function type procedural macro. Following that, lets create a custom macro.

```
// my_macro/src/lib.rs
extern crate proc_macro;

#[proc_macro]
pub fn make_answer(_item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}
```

But what does it do ?

To see what our custom proc macro expands to, lets use a utility: Cargo Expand. It allows us to see how our procedural macros expand. Then upon running `$ cargo expand` from the root of our project, we obtain:

```
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2021::*;
#[macro_use]
```

```
extern crate std;
extern crate proc_macro;
#[proc_macro]
pub fn make_answer(_item: proc_macro::TokenStream) -> proc_macro::TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}
const _: () = {
    extern crate proc_macro;
    #[rustc_proc_macro_decls]
    #[used]
    #[allow(deprecated)]
    static _DECLS: &[proc_macro::bridge::client::ProcMacro] = &[
        proc_macro::bridge::client::ProcMacro::bang("make_answer", make_answer),
    ];
};
```

A lot of text is added to our code, but most of it looks familiar like `#[prelude_import]` , `extern crate std;` . We would expect them to be there. But the interesting part is here:

```
static _DECLS: &[proc_macro::bridge::client::ProcMacro] = &[
    proc_macro::bridge::client::ProcMacro::bang("make_answer", make_answer),
];
```

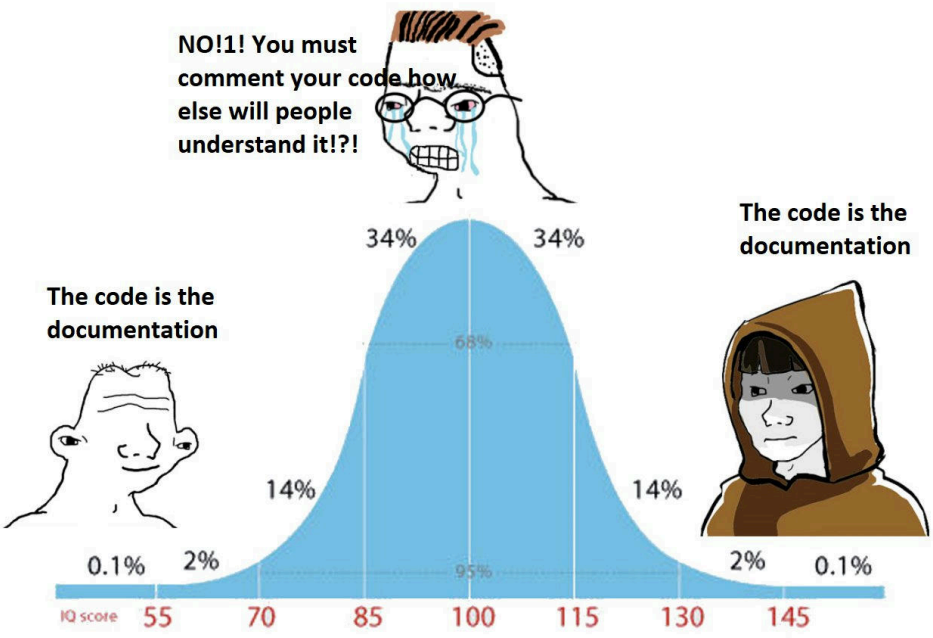
By just reading names, we see

- `proc_macro` : obviously,
- `bang` : name internally used for function type macros,
- `"make_answer"` : name of our proc macro.

Nothing surprising. But there is something new too: `bridge::client` .

At this point we have come deep enough. Our holy Rust Book does not tell us about the bridge. So lets try refering the Rustc Dev Guide → 36. Syntax and the AST → 36.2 Macro Expansion
I am telling you, the rust is quite well documented compared to most other language. **There is a book for everything!**

If you scroll **all** the way down, you might reach small section on Procedural Macros. But that does not matter to us. We are all chads here. We read the code.



So it tells us about `rustc_expand::proc_macro` and `rustc_expand::proc_macro_server` At this point we can piece together three words that we have come across:

1. `client`
2. `bridge`
3. `server`

This likey means that procedural macros work in a server-client architecture. And you would be correct!

§1.2. What is happening ?

This is a good place to explain how proc macros work internally.

Procedural Macros work on `TokenStream` . A TokenStream is just a stream of tokens. A token is just a group of character which have a collective meaning. For example if we take `let x = 2;` , we can say the tokens would look like:

let		x		=		2		;
Keyword	⟶	Variable Name	⟶	Logical Operator	⟶	Constant	⟶	Delimiter

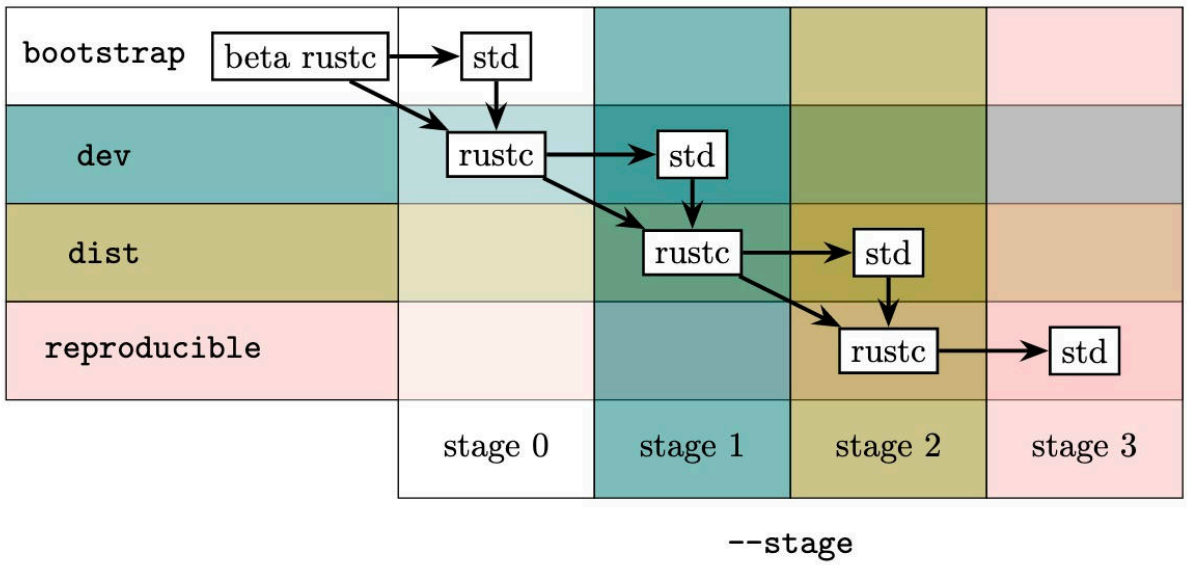
The names of tokens here is representive of logic rather than actual names used in compiler. The procedural macro takes in some `TokenStream` and outputs another `TokenStream` which replaces the original one. This “expansion” of the origial TokenStream happens at the compile time on the machine it is compiling on. Not during the runtime on the machine the code was built for. This is a unique problem while building the compiler.

§1.3. The Chicken and the Egg problem

1. **What came first, the Chicken or the Egg ?**
2. **When the first ever compiler was made, how did they compile it?**
3. **Can compiler compile code of compiler ?**

§1.3.1. Bootstrapping

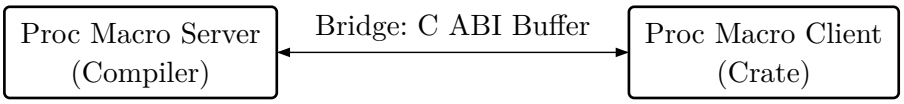
Bootstrapping is a technique for creating a self-compiling compiler, which is a compiler written in the same programming language it compiles. This is the same technique that the rust compiler uses. The best analogy I can think of is how the Terminator uses his left arm to heal his right arm. In similar fashion the rustc compiler and std library continuously build each other until we have the final output.



Read more at: Why is Rust's build system uniquely hard to use?

§1.3.2. Procedural Macro as part of library

Proc macros are also part of rust library. This means they have to be compatible between two different version of compiler. Therefore when the compiler calls the proc macro to run, the `TokenStream` is passed as **serializaed through a C ABI Buffer**. And thus the reaason proc macros use a server (compiler frontend) and client (proc macro client) architecture through a bridge (C ABI Buffer).



This also means that proc macro can not have depencency on any extern crate.

§1.4. Rustc_expand

Lets look at actual code for Rust's compiler. The entry point is rustc_expand::proc_macro. Here `fn expand` gets called for all 3 types of proc macros. This creates an instance of proc macro server defined at rustc_expand::proc_macro_server. Then the actual client being the proc macro crate is called through the proc_macro::bridge.

§2. Add Support for wasm proc macro

At this point we have explored all the words thats we discored through `$ cargo expand` . We understand overall structure and how pieces are interacting.

Problem 2.1.
But what about it ? All we want to do is add a way such that we can run proc macro written in rust.

§2.1. Compile Proc Macro to Wasm

Yes, so lets review. The first thing to run proc macro written in rust is to build proc macro to wasm.

Lets do that. For the macro we build earlier, run the command:

```
$ cargo build --target wasm32-unknown-unknown

# Output
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.06s
```

Voila! It builds!
But is there a `.wasm` file in `/target` ?

```
# Check yourself
$ ls **/*.wasm

#Output
Pattern, file or folder not found
```

No, none, nota, nill, null. What ?
Yes you can not build proc macros to `wasm` yet.
Currently this has been identified as lower on list of priorities and thus no work has been done.

§2.1.1. Current Work Around

Update your `lib.rs` file to:

```
// my_macro/src/lib.rs
extern crate proc_macro;

#[no_mangle]
#[export_name = "make_answer"]
pub extern "C" fn make_answer(_item: proc_macro::TokenStream) ->
proc_macro::TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}
```

Just compile `lib.rs` file to `wasm` using rustc.

```
$ rustc src/lib.rs --extern proc_macro --target wasm32-unknown-unknown --crate-type
lib
```

This has some glaring drawbacks as we will find later.

§2.2. Register a wasm crate

Now that we have our wasm file. Lets try using it how other proc macros are used.
If you already have some simple rust repo with a single proc macro dependency, you can try: `$ cargo build -vv` for super verbose output which will show us what it will do in the background which are just calls to the holy rust compiler **rustc**. You will see some stuff like:

```
   Compiling my-macro v0.1.0 (/Users/apurva/projects/proc-macro-server/my-macro)
        Running   rustc --crate-name my_macro --edition=2021 lib.rs --crate-type
proc-macro  -C prefer-dynamic -C embed-bitcode=no -C metadata=60c0b140b17fe75a -C
extra-filename=-60c0b140b17fe75a --out-dir /Users/apurva/projects/proc-macro-server/
run-macro/target/debug/deps -C incremental=/Users/apurva/projects/proc-macro-server/
run-macro/target/debug/incremental -L dependency=/Users/apurva/projects/proc-macro-
server/run-macro/target/debug/deps --extern proc_macro`
   Compiling run-macro v0.1.0 (/Users/apurva/projects/proc-macro-server/run-macro)
            Running   rustc   --crate-name   run_macro   --edition=2021   src/main.rs
--error-format=json   --json=diagnostic-rendered-ansi,artifacts,future-incompat   --
diagnostic-width=100  --crate-type  bin  --emit=dep-info,link  -C  embed-bitcode=no
-C  debuginfo=2  -C  split-debuginfo=unpacked  -C  metadata=3f481d1407db4a43  -C
extra-filename=-3f481d1407db4a43 --out-dir /Users/apurva/projects/proc-macro-server/
run-macro/target/debug/deps -C incremental=/Users/apurva/projects/proc-macro-server/
run-macro/target/debug/incremental -L dependency=/Users/apurva/projects/proc-macro-
server/run-macro/target/debug/deps  --extern  my_macro=/Users/apurva/projects/proc-
macro-server/run-macro/target/debug/deps/libmy_macro-60c0b140b17fe75a.dylib`
```

Too much garbage! I did not sign up for this.
Calm you horses buddy.
The first compilation just means it is building the proc macro. The second call for compiling is when it actually build the crate and attaches our macro using the line:

```
--extern my_macro=/some_file_path/libmy_macro-hash_for_incremental_comp.dylib
```

Along the same line lets try to use our wasm file by directly passing it through extern:

```
$ rustc /some_rust_file.rs --extern my_macro=/some_path/my_macro.wasm

# Output
(Some error)
```

Well we can not just pass wasm files to the compiler. Back to rust compiler dev guide!

Libraries and Metadata tells us that currently it only accepts 3 types of file

1. rlib
2. dylib
3. rmeta

So we need to also add wasm to this list. This is has been done but with a caveat. The `CrateLocator` works correctly and accepts a wasm file, however upon the next step we need to register the crate which requires metadata. Accoring to the guide (true by the way, :D):

"As crates are loaded, they are kept in the `CStore` with the crate metadata wrapped in the `CrateMetadata` struct."

We need `CrateMetadata` ! And currently while compiling wasm file, metadata is not attached to it. The glaring issue I told you about. Current hack is to just just patch it all with made up data.

§2.3. Expand a Proc Macro

Finally! The meat of the matter!

So now that we have registered the wasm file, we can use it to expand our proc macro. We already know which part of the compiler is reponsible: `rustc_expand::proc_macro` . Lets try to read **simplified** expand function for `BangProcMacro` . Read through the comments for small walkthrough.

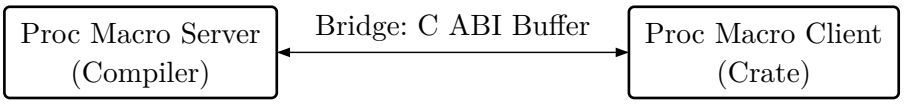
```
use rustc_ast::tokenstream::TokenStream;

impl base::BangProcMacro for BangProcMacro {
    fn expand<'cx>(
        ..
        // takes in stream of token defined by compiler
        input: TokenStream,

        //expects a result with new stream of tokens
    ) -> Result<TokenStream, ErrorGuaranteed> {
        ..
        // create instance of proc macro server
        let server = proc_macro_server::Rustc::new();
        // Run main entry function for proc macro
        // which takes care of talking between server and client
        // returns new tokenstream
        self.client.run(server, input, ..)
```

```
    }
}
```

Ok. Lets think again about what we want to do.

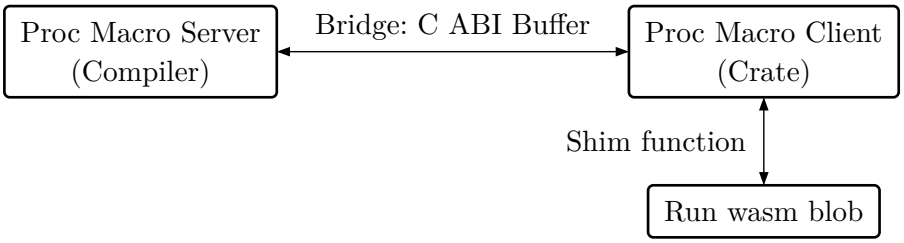


The only change to our above diagram is that now the **Proc Macro Client** is a wasm file. Which means we only need to change some logic for the client. So when do we create the client ? As hint again check the output of `$ cargo expand` .

This is the function used to create a new client:

```
impl Client<crate::TokenStream, crate::TokenStream> {
    pub const fn expand1(f: impl Fn(crate::TokenStream) -> crate::TokenStream + Copy)
-> Self {
        Client {
            get_handle_counters: HandleCounters::get,
            run: super::selfless_reify::reify_to_extern_c_fn_hrt_bridge(move |bridge|
{
                run_client(bridge, |input| f(crate::TokenStream(Some(input)))).0
            }),
            _marker: PhantomData,
        }
    }
}
```

This is not meant to make sense for you. The important part is that the function takes in our proc macro function and creates a client using it. The current leading implementation is to create a thin shim function for this input function which internally runs the wasm blob.



This looks like:

```
fn wasm_pm(ts: crate::TokenStream, path: PathBuf) -> crate::TokenStream {
    // call wasmtime using a shared library
    // and run the wasm blob internally
}
impl Client<crate::TokenStream, crate::TokenStream> {
    pub const fn expand_wasm(path: PathBuf) -> Self {
        let f = unsafe { wasm_pm };

        Client {
            get_handle_counters: HandleCounters::get,
            run: super::selfless_reify::reify_to_extern_c_fn_hrt_bridge(move |bridge|
{
                run_client(bridge, |input| f(crate::TokenStream(Some(input)), path).0)
```

```
            }),
            _marker: PhantomData,
        }
    }
}
```

§3. What is the current state of project ?

Ok mav, after reading through this for 10 hours, where are we at ?

I am at final stages of finishing getting the shim working. This has taken much longer than I personally expect. There can be many reasons:

1. Skill Issue
2. Skill Issue
3. Skill Issue
4. Skill Issue
5. Skill Issue

100. libproc_macro can not have dependency on any other crate. Which means every low level implementation has to be seperately defined and used for libproc_macro. So I have gone through more low level code than ever in life.

Look out for update on this soon.

After this, the efforts will be put into adding metadata when we compile proc macro to wasm and properly registering it as a crate.