Macro System WebAssembley

Hitesh Kumar Sharma
Instructor, Pluralsight

Agenda

- Macro
- Types of Macro
- WebAssembly
- Performance

Macro in Rust

- Macros provide functionality similar to functions but without the runtime cost. There is some compile-time cost, however, since macros are expanded during compile time.

- The macros in Rust are extremely dissimilar from the ones in C. Unlike C macros, which are used for text substitution, rust macros are applied to the token tree.

- Macros are well supported in Rust. Using macros, you can perform metaprogramming, which is the act of writing code that writes other code.

Types of Macro

Rust has two types of Macro:-

Declarative Macros: Declarative macros allow you create expressions that operate on the inputs you supply in the form of Rust code, just like a match expression does. It generates code to replace the macro invocation using the code you supply.

Procedural Macros: You can manipulate the given Rust code's abstract syntax tree (AST) by using procedural macros. A proc macro is a function that connects two TokenStreams, where the output takes the place of the macro invocation.

Declarative Macros

Declarative macros in Rust are defined using the macro_rules! macro. They follow the pattern-matching approach, where you define patterns and their corresponding replacement code. The basic syntax of a declarative macro is as follows:

```
macro_rules! macro_name {
    ( pattern1 ) => {
        // Replacement code for pattern1
    };
    ( pattern2 ) => {
        // Replacement code for pattern2
    };
    // More patterns and replacements...
}
```

- The macro_rules! keyword indicates the start of a macro definition.
- macro_name is the name of the macro you want to define.
- Each pattern is enclosed in parentheses ( ).
- After the =>, you write the replacement code for the matched pattern.

Declarative Macros (Example)

In this example, a simple declarative macro called greet, which will generate a simple greeting message with a name provided as an argument.

```
macro_rules! greet {
    ( $name:expr ) => {
        println!("Hello, {}!", $name);
    };
}
```

To use the greet macro, you call it with an argument that matches the defined pattern. For example:

```
fn main() {
    greet!("Alice");
    greet!("Bob");
}
```

In this example:

- The macro is named greet.
- The pattern ($name:expr) matches an expression (denoted by expr) and binds it to the variable $name.
- The replacement code println!("Hello, {}!", $name); will be executed when the macro is invoked with a valid pattern.

Procedural Macros (contd..)

There are three kinds of procedural macros in Rust:

1. **Custom Attribute Macros:** These macros define new attributes that you can apply to items in your code. The procedural macro will process the code corresponding to the attributed item and generate new code based on the attribute's behavior.
2. **Derive Macros:** Derive macros automatically implement traits for user-defined data structures. For example, you can use #[derive(Debug)] to automatically generate the Debug trait implementation for your structs.
3. **Function-Like Macros:** These macros take input tokens, perform transformations, and produce new tokens that are then used as part of the code.

Procedural Macros (Example)

Here's a simple example of a procedural macro crate that implements a custom derive macro to automatically implement a simple Show trait for structs:

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

use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, DeriveInput};

#[proc_macro_derive(Show)]
pub fn show_derive(input: TokenStream) ->
TokenStream {
    // Parse the input tokens into a
syntax tree
    let ast: DeriveInput =
parse_macro_input!(input);

// Get the name of the struct being
derived for
    let name = &ast.ident;
```

```
    // Generate the output code
implementing the Show trait
    let expanded = quote! {
        impl Show for #name {
            fn show(&self) -> String {
                format!("{:?}", self)
            }
        }
    };

    // Return the generated code as a
token stream
    TokenStream::from(expanded)
}
```

Procedural Macros (Example) (contd..)

With this procedural macro crate, users can now apply the #[derive(Show)] attribute to their structs, and the Show trait implementation will be automatically generated:

```
// In user's code
use my_derive::Show;

#[derive(Show)]
struct MyStruct {
    field1: i32,
    field2: String,
}

fn main() {
    let my_struct = MyStruct {
        field1: 42,
        field2: "Hello,
Rust!".to_string(),
    };
    println!("{}", my_struct.show());
}
```

When the user's code is compiled, the procedural macro will automatically generate the Show trait implementation for the MyStruct type, and the show() method can be called on instances of MyStruct.

WebAssembly in Rust

WebAssembly (Wasm) is a binary instruction format designed to run at near-native speed in web browsers. It allows you to execute code written in different programming languages in a secure and efficient manner directly within the browser environment. Rust is one of the programming languages that can be compiled to WebAssembly, allowing developers to write high-performance and safe code for the web.

To work with WebAssembly in Rust, you'll use the wasm32-unknown-unknown target, which is a target that doesn't rely on any underlying operating system or browser-specific APIs. It enables Rust code to be compiled to a WebAssembly binary. Here's how you can get started with WebAssembly in Rust:

Working of WebAssembly in Rust

- **Rust to WebAssembly Compilation:** Rust code can be compiled to WebAssembly using the wasm32-unknown-unknown target. This target ensures that Rust code is compiled without any dependencies on the underlying operating system or browser-specific APIs, making it suitable for running in various environments, including web browsers.

- **JavaScript Integration:** Once the Rust code is compiled to WebAssembly, it can be loaded and executed in a web page using JavaScript. JavaScript code can interact with WebAssembly modules through JavaScript APIs, allowing seamless integration with existing web applications and libraries.

Working of WebAssembly in Rust (Contd.)

3. Performance and Safety: Rust's focus on performance and safety carries over to WebAssembly. WebAssembly provides a sandboxed environment with predictable performance characteristics, and Rust's memory safety guarantees remain in effect, ensuring that WebAssembly modules are safe and secure.

4. Use Cases: WebAssembly in Rust opens up various possibilities for web development. It can be used to build performance-critical components, such as real-time graphics, audio processing, video decoding, and more. Additionally, WebAssembly allows developers to port existing Rust applications to the web, enabling code reuse and cross-platform compatibility.

Tools Required for WebAssembly in Rust

Install the required tools:

1. Install Rust: If you don't have Rust installed, you can download and install it from the official website (https://www.rust-lang.org/learn/get-started).

1. Install wasm-pack: wasm-pack is a command-line tool that helps you build and package Rust-generated WebAssembly projects. You can install it by following the instructions at https://rustwasm.github.io/wasm-pack/installer/.