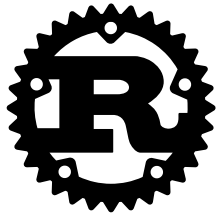
Rust 2019

compscicenter.ru

aleksey.kladov@gmail.com



Лекция 13
Макросы

Макросы

Способ абстракции над синтаксисом языка

- могущественный инструмент (свой синтаксис!)
- ограниченный инструмент (только синтаксис)

Макросы не успели доделать к 1.0, текущая версия не без недостатков :-)

Quick Tour

Macro By Example By Example

```
struct Function { ... }
struct Const { ... }
struct TypeAlias { ... }

pub enum TraitItem {
    Function(Function),
    Const(Const),
    TypeAlias(TypeAlias),
}
```

Poor man's OOP

Macro By Example By Example

```
impl From<Function> for TraitItem {
    fn from(item: Function) -> TraitItem {
        TraitItem::Function(item)
    }
}

impl From<Const> for TraitItem {
    ...
}

impl From<TypeAlias> for TraitItem {
    ...
}
```

Дублирование кода!

Macro By Example By Example

```
macro_rules! impl_froms {
    ($e:ident : $($v:ident),* ) => {
        $(
            impl From<$v> for $e {
                fn from(it: $v) -> $e { $e::$v(it) }
            }
        )*
    }
}

pub enum TraitItem {
    Function(Function),
    Const(Const),
    TypeAlias(TypeAlias),
}
impl_froms!(TraitItem: Function, Const, TypeAlias);
```

```
macro_rules! impl_froms {
    ($e:ident : $($v:ident),* ) => {
        $(
            impl From<$v> for $e {
                fn from(it: $v) -> $e { $e::$v(it) }
            }
        )*
    }
}
impl_froms!(TraitItem: Function, Const, TypeAlias);
```

- `impl_froms` — имя макроса
- `( ... ) =>` — паттерн
- `$e:ident` — макро переменная, матчит идентификатор
- `$($v:ident),*` — идентификаторы через `,`
- `=> { ... }` — результат раскрытия макроса

Token Trees

Синтаксис вызова макросов:

```
an::path! opt_name { token_tree }
```

`token_tree` — любая последовательность токенов Rust где `( )`, `[ ]` и `{ }` сбалансированы

Результат работы макроса — тоже token tree

Можно расширять синтаксис языка!

```
format!(
    "my name is {name}, my father's name is also {name}",
    name = "John"
);
```

`macro_rules` это тоже макрос!

```
macro_rules! impl_froms {
    ($e:ident : $($v:ident),* ) => {
        $(
            impl From<$v> for $e {
                fn from(it: $v) -> $e { $e::$v(it) }
            }
        )*
    }
}
```

Не нужно изобретать специальный синтаксис для конструкции языка

Примеры

`format!`, `println!`, `log::info!` — проверка количества аргументов без сложной системы типов

`vec!` — литерал коллекции без нового синтаксиса / vararg функций

`try!` — старая версия `?`: `try!(File::create("hello.txt"))`, отложили дизайн синтаксиса

Lazy language design!

Vs. C

В C макросы раскрываются препроцессором, а не компилятором

```c
#define squared(a) a * a

int main(void) {
    squared(1 + 1); // 1 + 1 * 1 + 1
    return 0;
}
```

Можно полностью поменять синтаксис языка

```c
#define begin {
#define end }
```

Vs. C

Rust сохраняет token trees

```
macro_rules! squared {
    ($e:expr) => { $e * $e }
}

fn main() {
    squared!(1 + 1); // [1 + 1] * [1 + 1]
}
```

Так как макросы синтаксически выделены (! + правильная скобочная последовательность), код на Rust можно парсить, не раскрывая макросов.

Vs. Scala

В Rust аргумент макроса — почти произвольная последовательность токенов

В Scala — выражение
⇒ синтаксических возможностей меньше, но IDE легче

В Rust раскрытие макросов — строго синтаксическое преобразование

В Scala макросы могут смотреть на типы
⇒ больше возможностей, но IDE тяжелее

Vs. Lisp

Дерево токенов — почти S-выражение

Можно преобразовывать деревья токенов произвольным кодом

Нельзя генерировать новый код во время исполнения (нет eval)

Macro By Example

Видимость

Совершенно другие правила видимости: макрос виден "после" объявления

```
// nop!(); // не работает
macro_rules! nop {
    () => ()
}
nop!();

mod m {
    nop!(); // Работает!
}

macro_rules! nop { // Переопределили макрос!
    () => { 92 }
}
```

Видимость

Макросы локальны для блоков

```
fn foo() -> i32 {
    macro_rules! bail {
        () => { return 92 }
    }

    if condition {
        bail!();
    }
}
// тут `bail` не видно
```

Видимость

Макросы локальны для модулей без `#[macro_use]`

```
mod foo {
    macro_rules! m1 { () => (()) }
}


#[macro_use]
mod bar {
    macro_rules! m2 { () => (()) }
}

// m2 виден, m1 нет
```

Видимость

При использовании макросов из других крейтов, обычные правила видимости

./log/src/lib.rs

```
#[macro_export]
macro_rules! info {
    ...
}
```

./main.rs

```
fn main() {
    log::info!("hello, {}", 92);
}
```

Паттерны

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

- item, expr, ty, vis …
-

$(...)*	$(...),*	$(...)+	$(...)?

Паттерны

```
macro_rules! info {
    (target: $target:expr, $($arg:tt)+) => (
        log!(target: $target, $crate::Level::Info, $($arg)+);
    );
    ($($arg:tt)+) => (
        log!($crate::Level::Info, $($arg)+);
    );
}
```

- можно перечислить несколько паттернов через `;`
- вид скобочек не имеет значение
- макросы могут быть рекурсивными
- всё, что не $, сопоставляется буквально
- **`$crate`** ?

Гигиена

Макросы частично гигиеничные

```
macro_rules! declare_x {
    () => {
        let x = 92;
    }
}

fn main() {
    let x = 62;
    declare_x!();
    println!("{}", x); // 62
}
```

Гигиена

Можно передать идентификатор в макрос

```
macro_rules! declare_var {
    ($var:ident) => {
        let $var = 92;
    }
}

fn main() {
    let x = 62;
    declare_var!(x);
    println!("{}", x); // 92
}
```

Гигиена

Гигиена работает не везде :-(

```
macro_rules! declare_fn {
    () => {
        fn foo() {}
    }
}

declare_fn!();

fn main() {
    foo(); // Ok :-(
}
```

Грабли

Гигиена + token trees решают часть проблем макросов, но не все

```
macro_rules! min {
    ($x:expr, $y:expr) => {
        if $x < $y { $x } else { $y }
    }
}
```

В чём тут проблема?

Грабли

$x может быть вычислен дважды!

```
macro_rules! min {
    ($x:expr, $y:expr) => {
        match ($x, $y) {
            (x, y) => if x < y { x } else { y }
        }
    }
}
```

Грабли

my-crate

```
#[macro_export]
macro_rules! foo {
    () => {
        use crate::X;
        ...
    }
}
```

other-crate

```
use my_crate::foo;

fn main() {
    foo!(); // crate::X будет указывать на other_crate
}
```

Грабли

my-crate

```
#[macro_export]
macro_rules! foo {
    () => {
        use $crate::X;
        ...
    }
}
```

other-crate

```
use my_crate::foo;

fn main() {
    foo!();
}
```

Встроенные Макросы

- `dbg!` — напечатать и вернут значение выражения:
  `dbg!(foo.bar()).baz`
- `include_str!` / `include_bytes!` — вставить ресурс в бинарь:
  `const LOGO: &[u8] = include_bytes!("assets/logo.png")`
- `file!` / `line!` / `column!` — имя файла, текущая строка, позиция
- `stringify!` — превращает аргумент в строку
- `format_args!` — главный `prlint`-style макрос, превращает `"foo = {}", foo` в `FormatArgs<'a>`

Условная Компиляция

```rust
#[cfg(unix)]
pub fn bytes2path(bytes: &[u8]) -> CargoResult<PathBuf> {
    use std::os::unix::prelude::*;
    Ok(PathBuf::from(OsStr::from_bytes(bytes)))
}

#[cfg(windows)]
pub fn bytes2path(bytes: &[u8]) -> CargoResult<PathBuf> {
    use std::str;
    match str::from_utf8(bytes) {
        Ok(s) => Ok(PathBuf::from(s)),
        Err(..) => Err(failure::format_err!(
            "invalid non-unicode path"
        )),
    }
}
```

Атрибут `cfg` скрывает неактивные определения. Условная компиляция — синтаксическая, на этапе раскрытия макросов :(

Условная Компиляция

Стандартный паттерн написание unit-тестов:

```
#[cfg(test)]
mod tests {
    use some_lib::test_specific_function;
    use super::*;

    #[test]
    fn test_foo() { ... }
}
```

- `#[test]` никогда не попадают в реальный бинарь/библиотеку
- `#[cfg(test)]` позволяет сгруппировать тесты и избавится от unused import

Условная Компиляция

```rust
pub fn dylib_path_envvar() -> &'static str {
    if cfg!(windows) {
        "PATH"
    } else if cfg!(target_os = "macos") {
        "DYLD_FALLBACK_LIBRARY_PATH"
    } else {
        "LD_LIBRARY_PATH"
    }
}
```

Макрос `cfg!` можно использовать в выражениях

Условная Компиляция

```
fn main() {
    let compile_time_path = env!("PATH");
    println!(
        "PATH at *compile* time:\n{}",
        compile_time_path,
    );
}
```

Макросы `env!` и `option_env!` позволяют смотреть на переменные окружения в compile time

Процедурные Макросы

Макросами могут быть обычные функции:

```
pub enum TokenTree {
    Group(Group),
    Ident(Ident),
    Punct(Punct),
    Literal(Literal),
}

impl Group {
    pub fn delimiter(&self) -> Delimiter
    pub fn stream(&self) -> TokenStream
}

type TokenStream = Iterator<Item = TokenTree>; // 👋

#[proc_macro]
pub fn squared(arg: TokenStream) -> TokenStream {
    format!("({arg}) * ({arg})", arg = arg).parse().unwrap()
}
```

Процедурные Макросы

- нет гигиены
- можно использовать только на уровне модуля (пример со `squared` не работает)
- нужно опредеять в отдельном крейте с

```
[lib]
proc-macro = true
```

Derive

```
#[derive(Clone, Copy, PartialEq, Eq)]
struct Vec3([f3; 3]);
```

`derive` это тоже макрос, `derive(Clone)` работает синтаксически

Можно писать свои `derive`!

```
#[proc_macro_derive(MyTrait)]
pub fn derive_my_trait(input: TokenStream) -> TokenStream {
    ...
}
```

Derive

Для `Derive`, нужно распарсить дерево токенов как определение ADT

Крейт `syn` позволяет парсить деревья токенов в AST Rust

`syn` — обычный код, никак не связанный с компилятором

Пример

Хотя интерфейс процедурных макросов простой, API syn очень большое!

```
#[derive(HeapSize)]
struct Demo<'a, T: ?Sized> {
    a: Box<T>,
    b: u8,
    c: &'a str,
    d: String,
}
```

`HeapSize` — размер объекта, учитывая данные в куче

"Глубокий" `std::size_of`

```rust
#[proc_macro_derive(HeapSize)]
pub fn derive_heap_size(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    let name = input.ident;
    let generics = add_trait_bounds(input.generics);
    let (impl_generics, ty_generics, where_clause) =
        generics.split_for_impl();
    let sum = heap_size_sum(&input.data);
    let expanded = quote! {
        impl #impl_generics heapsize::HeapSize
        for #name #ty_generics
            #where_clause
        {
            fn heap_size_of_children(&self) -> usize {
                #sum
            }
        }
    };
    proc_macro::TokenStream::from(expanded)
}
```

```
fn heap_size_sum(data: &Data) -> TokenStream {
    match *data {
        Data::Struct(ref data) => {
            match data.fields {
                Fields::Unnamed(ref fields) => {
                    let recurse = fields.unnamed.iter()
                        .enumerate().map(|(i, f)| {
                            let index = Index::from(i);
                            quote! {
                                HeapSize::heap_size_of_children(
                                    &self.#index
                                )
                            }
                        });
                    quote! { 0 #(+ #recurse)* }
                }
                Fields::Named(_) | Fields::Unit => panic!("TODO")
            }
        }
        Data::Enum(_) | Data::Union(_) => panic!("TODO")
    }
}
```

serde

https://serde.rs/

serde

Фреймворк для сериализации и десериализации на Rust

Один из самый замечательных крейтов!

Трейты + Макросы = гибкая и быстрая сериализация вне языка

serde

Ядро дизайна — статически диспетчеризуемый visitor

```
#[derive(Serialize)]
struct Rgb {
    r: u8,
    g: u8,
    b: u8,
}
```

```rust
struct Rgb {
    r: u8,
    g: u8,
    b: u8,
}

impl Serialize for Rgb {
  fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
  where
      S: Serializer,
  {
      let mut rgb = serializer.serialize_struct("Rgb", 3)?;
      rgb.serialize_field("r", &self.r)?;
      rgb.serialize_field("g", &self.g)?;
      rgb.serialize_field("b", &self.b)?;
      rgb.end()
  }
}
```

Статический reflection!

Serialize

```rust
pub trait Serialize {
  fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
  where
      S: Serializer;
}
```

S — формат данных (JSON, YAML, XML)

S — параметр типа, после мономорфизации получаем код, который напрямую создаёт JSON

Serializer

```
pub trait Serializer: Sized {
    type Ok;
    type Error: Error;

    type SerializeStruct
        : SerializeStruct<Ok = Self::Ok, Error = Self::Error>;

    fn serialize_i8(self, v: i8)
        -> Result<Self::Ok, Self::Error>;

    fn serialize_i16(self, v: i16)
        -> Result<Self::Ok, Self::Error>;

    fn serialize_struct(
        self,
        name: &'static str,
        len: usize,
    ) -> Result<Self::SerializeStruct, Self::Error>;
}
```


FIN
DE
CINEMA