Бесплатная электронная книга

учусь Rust

Free unaffiliated eBook created from
Stack Overflow contributors.

#rust

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rust

It is an unofficial and free Rust ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Rust.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Rust

замечания

Rust - это язык системного программирования, предназначенный для обеспечения безопасности, скорости и параллелизма. Rust имеет множество функций времени и безопасности во время компиляции, чтобы избежать сбоев данных и общих ошибок, все с минимальными издержками до нуля.

Версии

стабильный

Версия	Дата выхода
1.17.0	2017-04-27
1.16.0	2017-03-16
1.15.1	2017-02-09
1.15.0	2017-02-02
1.14.0	2016-12-22
1.13.0	2016-11-10
1.12.1	2016-10-20
1.12.0	2016-09-30
1.11.0	2016-08-18
1.10.0	2016-07-07
1.9.0	2016-05-26
1.8.0	2016-04-14
1.7.0	2016-03-03
1.6.0	2016-01-21
1.5.0	2015-12-10
1.4.0	2015-10-29

Версия	Дата выхода
1.3.0	2015-09-17
1.2.0	2015-08-07
1.1.0	2015-06-25
1.0.0	2015-05-15

Бета

Версия	Ожидаемая дата выпуска
1.18.0	2017-06-08

Examples

Расширенное использование println!

`println!` (и его брат, `print!`) обеспечивает удобный механизм для создания и печати текста, который содержит динамические данные, аналогичные семейству функций `printf` найденным на многих других языках. Его первым аргументом является *строка формата* , которая определяет, как другие аргументы должны быть напечатаны как текст. Строка формата может содержать заполнители (заключенные в `{}`), чтобы указать, что должна произойти замена:

```
// No substitution -- the simplest kind of format string
println!("Hello World");
// Output: Hello World

// The first {} is substituted with a textual representation of
// the first argument following the format string. The second {}
// is substituted with the second argument, and so on.
println!("{} {} {}", "Hello", true, 42);
// Output: Hello true 42
```

На этом этапе вы можете спросить: как `println!` знать, чтобы напечатать логическое значение `true` как строку «true»? `{}` действительно является инструкцией для форматирования, что значение должно быть преобразовано в текст, используя показатель `Display` . Эта черта реализована для большинства примитивных типов ржавчины (строки, числа, булевы и т. Д.) И предназначена для «пользовательского вывода». Следовательно, число 42 будет напечатано в десятичном виде как 42, а не, скажем, в двоичном виде, то есть как оно хранится внутри.

Как мы печатаем типы, которые *не* реализуют `Display` , примеры - `[i32]` (`[i32]`), векторы (

`Vec<i32>`) или опции (`Option<&str>`)? Нет четкого пользовательского текстового представления (т. Е. Вы можете тривиально вставлять в предложение). Чтобы облегчить печать таких значений, Rust также имеет свойство `Debug` и соответствующий `{:?}` Placeholder. Из документации: « `Debug` должна форматировать вывод в контексте программирования, отладочном контексте». Давайте посмотрим несколько примеров:

```
println!("{:?}", vec!["a", "b", "c"]);
// Output: ["a", "b", "c"]

println!("{:?}", Some("fantastic"));
// Output: Some("fantastic")

println!("{:?}", "Hello");
// Output: "Hello"
// Notice the quotation marks around "Hello" that indicate
// that a string was printed.
```

`Debug` также имеет встроенный механизм довольно печати, который вы можете включить с помощью модификатора `#` после двоеточия:

```
println!("{:#?}", vec![Some("Hello"), None, Some("World")]);
// Output: [
//     Some(
//         "Hello"
//     ),
//     None,
//     Some(
//         "World"
//     )
// ]
```

Строки формата позволяют выразить довольно сложные замены :

```
// You can specify the position of arguments using numerical indexes.
println!("{1} {0}", "World", "Hello");
// Output: Hello World

// You can use named arguments with format
println!("{greeting} {who}!", greeting="Hello", who="World");
// Output: Hello World

// You can mix Debug and Display prints:
println!("{greeting} {1:?}, {0}", "and welcome", Some(42), greeting="Hello");
// Output: Hello Some(42), and welcome
```

`println!` и друзья также предупреждают вас, если вы пытаетесь сделать что-то, что не будет работать, а не сбой во время выполнения:

```
// This does not compile, since we don't use the second argument.
println!("{}", "Hello World", "ignored");

// This does not compile, since we don't give the second argument.
println!("{} {}", "Hello");
```

```
// This does not compile, since Option type does not implement Display
println!("{}", Some(42));
```

По своей сути макросы печати Rust - это просто обертки вокруг `format!` макрос, который позволяет создавать строку путем сшивания текстовых представлений разных значений данных. Таким образом, для всех приведенных выше примеров вы можете заменить `println!` для `format!` для сохранения форматированной строки вместо ее печати:

```
let x: String = format!("{} {}", "Hello", 42);
assert_eq!(x, "Hello 42");
```

Консольный вывод без макросов

```
// use Write trait that contains write() function
use std::io::Write;

fn main() {
    std::io::stdout().write(b"Hello, world!\n").unwrap();
}
```

- `std::io::Write` trait предназначен для объектов, которые принимают потоки байтов. В этом случае дескриптор стандартного вывода получается с помощью `std::io::stdout()` .
- `Write::write()` принимает байтовый срез (`&[u8]`), который создается с помощью литерала с байтовой строкой (`b"<string>"`). `Write::write()` возвращает `Result<usize, IoError>` , который содержит либо количество записанных байтов (по успеху), либо значение ошибки (при сбое).
- Вызов `Result::unwrap()` указывает, что ожидается, что вызов будет успешным (`Result<usize, IoError> -> usize`), и значение будет отброшено.

Минимальный пример

Чтобы написать традиционную программу Hello World в Rust, создайте текстовый файл `hello.rs` содержащий следующий исходный код:

```
fn main() {
    println!("Hello World!");
}
```

Это определяет новую функцию `main` , которая не принимает никаких параметров и не возвращает никаких данных. Здесь ваша программа запускается при запуске. Внутри этого у вас есть `println!` , который представляет собой макрос, который печатает текст в консоли.

Чтобы сгенерировать двоичное приложение, вызовите компилятор Rust, передав ему имя

исходного файла:

```
$ rustc hello.rs
```

Получаемый исполняемый файл будет иметь то же имя, что и основной исходный модуль, поэтому для запуска программы в системе Linux или MacOS выполните:

```
$ ./hello
Hello World!
```

В системе Windows запустите:

```
C:\Rust> hello.exe
Hello World!
```

Начиная

Установка

Прежде чем вы сможете что-либо использовать с использованием языка программирования Rust, вам нужно будет его приобрести - либо для Windows, либо используя ваш терминал в *Unix-подобных* системах, где `$` символизирует вход в терминал:

```
$ curl https://sh.rustup.rs -sSf | sh
```

Это приведет к извлечению необходимых файлов и настройке последней версии Rust для вас, независимо от того, в какой системе вы находитесь. Для получения дополнительной информации см. Страницу проекта .

Примечание. Некоторые дистрибутивы Linux (например, Arch Linux) предоставляют `rustup` как пакет, который можно установить вместо него. И хотя многие Unix-подобные системы обеспечивают `rustc` и `cargo` как отдельные пакеты, по-прежнему рекомендуется использовать `rustup` поскольку это упрощает управление несколькими каналами выпуска и выполняет кросс-компиляцию.

Компилятор ржавчины

Теперь мы можем проверить, действительно ли *Rust* был успешно установлен на наших компьютерах, выполнив следующую команду либо в нашем терминале - если в UNIX - или в командной строке - если в Windows:

```
$ rustc --version
```

Если эта команда будет успешной, версия компилятора *Rust* , установленная на наших компьютерах, будет отображаться на наших глазах.

грузовой

С Rust приходит *Cargo* , который является инструментом построения, который используется для управления пакетами и проектами *Rust* . Чтобы убедиться, что это тоже присутствует на вашем компьютере, запустите в консоли консоль, ссылаясь на терминал или командную строку, в зависимости от того, в какой системе вы находитесь:

```
$ cargo --version
```

Как и эквивалентная команда для компилятора *Rust* , это вернет и отобразит текущую версию *Cargo* .

Чтобы создать свой первый проект Cargo, вы можете отправиться в Cargo .

Кроме того, вы можете скомпилировать программы напрямую с помощью `rustc` как показано в минимальном примере .

Прочитайте Начало работы с Rust онлайн: https://riptutorial.com/ru/rust/topic/362/начало-работы-с-rust

глава 2: Loops

Синтаксис

- loop { *block* } // бесконечный цикл
- a *условие* { *block* }
- a пусть *pattern* = *expr* { *block* }
- для *шаблона* в *expr* { *block* } // expr должен реализовать IntoIterator
- continue // перейти к концу тела цикла, при необходимости начать новую итерацию
- break // остановить цикл
- ' *label* : loop { *block* }
- ' *label* : while *условие* { *block* }
- ' *label* : while let *pattern* = *expr* { *block* }
- ' *label* : для *шаблона* в *expr* { *block* }
- продолжить *'метки* // переход к концу меченой *этикетки* тела цикла, начиная с новой итерацией в случае необходимости
- *метка* BREAK '// остановить *метку* цикла меченого

Examples

ОСНОВЫ

В Rust есть 4 петлевые конструкции. Все приведенные ниже примеры дают одинаковый результат.

Бесконечные петли

```
let mut x = 0;
loop {
    if x > 3 { break; }
    println!("{}", x);
    x += 1;
}
```

В то время как петли

```
let mut x = 0;
while x <= 3 {
    println!("{}", x);
    x += 1;
}
```

Также смотрите: В чем разница между `loop` и `while true` ?

Циклы, совпадающие с шаблонами

Они иногда известны как `while let` петлям для краткости.

```
let mut x = Some(0);
while let Some(v) = x {
    println!("{}", v);
    x = if v < 3 { Some(v + 1) }
        else     { None };
}
```

Это эквивалентно `match` внутри блока `loop` :

```
let mut x = Some(0);
loop {
    match x {
        Some(v) => {
            println!("{}", v);
            x = if v < 3 { Some(v + 1) }
                else     { None };
        }
        _       => break,
    }
}
```

Для циклов

В Rust `for` цикла может использоваться только с «итерируемым» объектом (т.е. он должен реализовывать `IntoIterator`).

```
for x in 0..4 {
    println!("{}", x);
}
```

Это эквивалентно следующему фрагменту, включая `while let` :

```
let mut iter = (0..4).into_iter();
```

```
while let Some(v) = iter.next() {
    println!("{}", v);
}
```

Примечание: `0..4` возвращает объект `Range` который уже реализует свойство `Iterator` . Поэтому `into_iter()` не является необходимым, но сохраняется только для иллюстрации того, что `for` делает. `IntoIterator` обзор см. В официальных документах `for` циклов и `IntoIterator` .

См. Также: Итераторы

Подробнее о циклах

Как уже упоминалось в «Основах», мы можем использовать все, что реализует `IntoIterator` с циклом `for` :

```
let vector = vec!["foo", "bar", "baz"]; // vectors implement IntoIterator
for val in vector {
    println!("{}", val);
}
```

Ожидаемый результат:

```
foo
bar
baz
```

Обратите внимание, что итерация по `vector` таким образом потребляет его (после цикла `for` `vector` не может использоваться снова). Это связано с тем, что `IntoIterator::into_iter` движется `self` .

`IntoIterator` также реализуется с помощью `&Vec<T>` и `&mut Vec<T>` (выводя значения с типами `&T` и `&mut T` соответственно), поэтому вы можете предотвратить перемещение `vector` , просто передав его по ссылке:

```
let vector = vec!["foo", "bar", "baz"];
for val in &vector {
    println!("{}", val);
}
println!("{:?}", vector);
```

Заметим, что `val` имеет тип `&&str` , так как `vector` имеет тип `Vec<&str>` .

Управление петлей

Все петлевые конструкции позволяют использовать выражения `break` и `continue` . Они воздействуют на ближайшую окружную (самую внутреннюю) петлю.

Управление базовым контуром

`break` завершает цикл:

```
for x in 0..5 {
    if x > 2 { break; }
    println!("{}", x);
}
```

Выход

```
0
1
2
```

`continue` завершение текущей итерации раньше

```
for x in 0..5 {
    if x < 2 { continue; }
    println!("{}", x);
}
```

Выход

```
2
3
4
```

Управление расширенным контуром

Теперь предположим, что у нас есть вложенные петли и мы хотим `break` во внешний цикл. Затем мы можем использовать метки меток, чтобы указать, к какому циклу относится `break` или `continue` . В следующем примере `'outer` - это метка, присвоенная внешнему циклу.

```
'outer: for i in 0..4 {
    for j in i..i+2 {
        println!("{} {}", i, j);
        if i > 1 {
            continue 'outer;
        }
    }
    println!("--");
}
```

Выход

```
0 0
0 1
--
```

```
1 1
1 2
--
2 2
3 3
```

При `i > 1` внутренний цикл повторялся только один раз и `--` не печатался.

Примечание. Не путайте метку цикла с переменной времени жизни. Постоянные переменные встречаются только рядом с `&` или как общий параметр внутри `<>` .

Прочитайте Loops онлайн: https://riptutorial.com/ru/rust/topic/955/loops

глава 3: PhantomData

Examples

Использование PhantomData в качестве маркера типа

Использование типа `PhantomData` подобно этому позволяет вам использовать определенный тип, не требуя, чтобы он был частью Struct.

```
use std::marker::PhantomData;

struct Authenticator<T: GetInstance> {
    _marker: PhantomData<*const T>, // Using `*const T` indicates that we do not own a T
}

impl<T: GetInstance> Authenticator<T> {
    fn new() -> Authenticator<T> {
        Authenticator {
            _marker: PhantomData,
        }
    }

    fn auth(&self, id: i64) -> bool {
        T::get_instance(id).is_some()
    }
}

trait GetInstance {
    type Output; // Using nightly this could be defaulted to `Self`
    fn get_instance(id: i64) -> Option<Self::Output>;
}

struct Foo;

impl GetInstance for Foo {
    type Output = Self;
    fn get_instance(id: i64) -> Option<Foo> {
        // Here you could do something like a Database lookup or similarly
        if id == 1 {
            Some(Foo)
        } else {
            None
        }
    }
}

struct User;

impl GetInstance for User {
    type Output = Self;
    fn get_instance(id: i64) -> Option<User> {
        // Here you could do something like a Database lookup or similarly
        if id == 2 {
            Some(User)
        } else {
```

```
            None
        }
    }
}

fn main() {
    let user_auth = Authenticator::<User>::new();
    let other_auth = Authenticator::<Foo>::new();

    assert!(user_auth.auth(2));
    assert!(!user_auth.auth(1));

    assert!(other_auth.auth(1));
    assert!(!other_auth.auth(2));

}
```

Прочитайте PhantomData онлайн: https://riptutorial.com/ru/rust/topic/7226/phantomdata

глава 4: Regex

Вступление

Стандартная библиотека Rust не содержит никакого парсера / совпадения регулярных выражений, но ящик `regex` (который находится в ржавчине-питомнике и, следовательно, полуофициальном) предоставляет парсер регулярных выражений. В этом разделе документации будет приведен краткий обзор использования ящика `regex` в обычных ситуациях, а также инструкции по установке и любые другие полезные замечания, которые необходимы при использовании ящика.

Examples

Простое совпадение и поиск

Регулярная поддержка выражения для подталкивания обеспечивается ящиком `regex` , добавляет его в ваш `Cargo.toml` :

```
[dependencies]
regex = "0.1"
```

Основным интерфейсом ящика `regex::Regex regex` является `regex::Regex` :

```
extern crate regex;
use regex::Regex;

fn main() {
    //"r" stands for "raw" strings, you probably
    // need them because rustc checks escape sequences,
    // although you can always use "\\" withour "r"
    let num_regex = Regex::new(r"\d+").unwrap();
    // is_match checks if string matches the pattern
    assert!(num_regex.is_match("some string with number 1"));

    let example_string = "some 123 numbers";
    // Regex::find searches for pattern and returns Option<(usize,usize)>,
    // which is either indexes of first and last bytes of match
    // or "None" if nothing matched
    match num_regex.find(example_string) {
        // Get the match slice from string, prints "123"
        Some(x) => println!("{}", &example_string[x.0 .. x.1]),
        None    => unreachable!()
    }
}
```

Группы захвата

```
extern crate regex;
```

```
use regex::Regex;

fn main() {
    let rg = Regex::new(r"was (\d+)").unwrap();
    // Regex::captures returns Option<Captures>,
    // first element is the full match and others
    // are capture groups
    match rg.captures("The year was 2016") {
        // Access captures groups via Captures::at
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.at(1)),
        None    => unreachable!()
    }

    // Regex::captures also supports named capture groups
    let rg_w_named = Regex::new(r"was (?P<year>\d+)").unwrap();
    match rg_w_named.captures("The year was 2016") {
        // Named capures groups are accessed via Captures::name
        // Prints Some("2016")
        Some(x) => println!("{:?}", x.name("year")),
        None    => unreachable!()
    }

}
```

Замена

```
extern crate regex;
use regex::Regex;

fn main() {
    let rg = Regex::new(r"(\d+)").unwrap();

    // Regex::replace replaces first match
    // from it's first argument with the second argument
    // => Some string with numbers (not really)
    rg.replace("Some string with numbers 123", "(not really)");

    // Capture groups can be accesed via $number
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $1)");

    let rg = Regex::new(r"(?P<num>\d+)").unwrap();

    // Named capture groups can be accesed via $name
    // => Some string with numbers (which are 123)
    rg.replace("Some string with numbers 123", "(which are $num)");

    // Regex::replace_all replaces all the matches, not only the first
    // => Some string with numbers (not really) (not really)
    rg.replace_all("Some string with numbers 123 321", "(not really)");
}
```

Прочитайте Regex онлайн: https://riptutorial.com/ru/rust/topic/7184/regex

глава 5: rustup

Вступление

`rustup` управляет вашей установкой ржавчины и позволяет устанавливать разные версии, которые можно легко настроить и легко обменивать.

Examples

Настройка

Установите инструментальную цепочку с помощью

```
curl https://sh.rustup.rs -sSf | sh
```

У вас должна быть последняя стабильная версия ржавчины. Вы можете проверить это, набрав

```
rustc --version
```

Если вы хотите обновить, просто запустите

```
rustup update
```

Прочитайте rustup онлайн: https://riptutorial.com/ru/rust/topic/8942/rustup

глава 6: Serde

Вступление

Serde является популярным **сер** ialization и **де** сериализации рамки Русте, используемый для преобразования в *последовательную* форму *данных* (например , JSON и XML) для Rust структур , и наоборот. Serde поддерживает множество форматов, включая: JSON, YAML, TOML, BSON, Pickle и XML.

Examples

Struct ↔ JSON

main.rs

```
extern crate serde;
extern crate serde_json;

// Import this crate to derive the Serialize and Deserialize traits.
#[macro_use] extern crate serde_derive;

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a packed JSON string. To convert it to
    // pretty JSON with indentation, use `to_string_pretty` instead.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Cargo.toml

```
[package]
```

```
name = "serde-example"
version = "0.1.0"
build = "build.rs"

[dependencies]
serde = "0.9"
serde_json = "0.9"
serde_derive = "0.9"
```

Сериализовать enum как строку

```
extern crate serde;
extern crate serde_json;

macro_rules! enum_str {
    ($name:ident { $($variant:ident($str:expr), )* }) => {
        #[derive(Clone, Copy, Debug, Eq, PartialEq)]
        pub enum $name {
            $($variant,)*
        }

        impl ::serde::Serialize for $name {
            fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
                where S: ::serde::Serializer,
            {
                // Serialize the enum as a string.
                serializer.serialize_str(match *self {
                    $( $name::$variant => $str, )*
                })
            }
        }

        impl ::serde::Deserialize for $name {
            fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
                where D: ::serde::Deserializer,
            {
                struct Visitor;

                impl ::serde::de::Visitor for Visitor {
                    type Value = $name;

                    fn expecting(&self, formatter: &mut ::std::fmt::Formatter) ->
::std::fmt::Result {
                        write!(formatter, "a string for {}", stringify!($name))
                    }

                    fn visit_str<E>(self, value: &str) -> Result<$name, E>
                        where E: ::serde::de::Error,
                    {
                        match value {
                            $( $str => Ok($name::$variant), )*
                            _ => Err(E::invalid_value(::serde::de::Unexpected::Other(
                                &format!("unknown {} variant: {}", stringify!($name), value)
                            ), &self)),
                        }
                    }
                }

                // Deserialize the enum from a string.
```

```
                deserializer.deserialize_str(Visitor)
            }
        }
    }
}

enum_str!(LanguageCode {
    English("en"),
    Spanish("es"),
    Italian("it"),
    Japanese("ja"),
    Chinese("zh"),
});

fn main() {
    use LanguageCode::*;
    let languages = vec![English, Spanish, Italian, Japanese, Chinese];

    // Prints ["en","es","it","ja","zh"]
    println!("{}", serde_json::to_string(&languages).unwrap());

    let input = r#" "ja" "#;
    assert_eq!(Japanese, serde_json::from_str(input).unwrap());
}
```

Сериализовать поля как camelCase

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

#[derive(Serialize)]
struct Person {
    #[serde(rename="firstName")]
    first_name: String,
    #[serde(rename="lastName")]
    last_name: String,
}

fn main() {
    let person = Person {
        first_name: "Joel".to_string(),
        last_name: "Spolsky".to_string(),
    };

    let json = serde_json::to_string_pretty(&person).unwrap();

    // Prints:
    //
    //    {
    //      "firstName": "Joel",
    //      "lastName": "Spolsky"
    //    }
    println!("{}", json);
}
```

Значение по умолчанию для поля

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

#[derive(Deserialize, Debug)]
struct Request {
    // Use the result of a function as the default if "resource" is
    // not included in the input.
    #[serde(default="default_resource")]
    resource: String,

    // Use the type's implementation of std::default::Default if
    // "timeout" is not included in the input.
    #[serde(default)]
    timeout: Timeout,

    // Use a method from the type as the default if "priority" is not
    // included in the input. This may also be a trait method.
    #[serde(default="Priority::lowest")]
    priority: Priority,
}

fn default_resource() -> String {
    "/".to_string()
}

/// Timeout in seconds.
#[derive(Deserialize, Debug)]
struct Timeout(u32);
impl Default for Timeout {
    fn default() -> Self {
        Timeout(30)
    }
}

#[derive(Deserialize, Debug)]
enum Priority { ExtraHigh, High, Normal, Low, ExtraLow }
impl Priority {
    fn lowest() -> Self { Priority::ExtraLow }
}

fn main() {
    let json = r#"
        [
          {
            "resource": "/users"
          },
          {
            "timeout": 5,
            "priority": "High"
          }
        ]
    "#;

    let requests: Vec<Request> = serde_json::from_str(json).unwrap();

    // The first request has resource="/users", timeout=30, priority=ExtraLow
    println!("{:?}", requests[0]);

    // The second request has resource="/", timeout=5, priority=High
    println!("{:?}", requests[1]);
```

```
}
```

Пропустить поле сериализации

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

use std::collections::BTreeMap as Map;

#[derive(Serialize)]
struct Resource {
    // Always serialized.
    name: String,

    // Never serialized.
    #[serde(skip_serializing)]
    hash: String,

    // Use a method to decide whether the field should be skipped.
    #[serde(skip_serializing_if="Map::is_empty")]
    metadata: Map<String, String>,
}

fn main() {
    let resources = vec![
        Resource {
            name: "Stack Overflow".to_string(),
            hash: "b6469c3f31653d281bbbfa6f94d60fea130abe38".to_string(),
            metadata: Map::new(),
        },
        Resource {
            name: "GitHub".to_string(),
            hash: "5cb7a0c47e53854cd00e1a968de5abce1c124601".to_string(),
            metadata: {
                let mut metadata = Map::new();
                metadata.insert("headquarters".to_string(),
                                "San Francisco".to_string());
                metadata
            },
        },
    ];

    let json = serde_json::to_string_pretty(&resources).unwrap();

    // Prints:
    //
    //    [
    //      {
    //        "name": "Stack Overflow"
    //      },
    //      {
    //        "name": "GitHub",
    //        "metadata": {
    //          "headquarters": "San Francisco"
    //        }
    //      }
    //    ]
    println!("{}", json);
```

```
}
```

Внедрить Serialize для пользовательского типа карты

```
impl<K, V> Serialize for MyMap<K, V>
    where K: Serialize,
          V: Serialize
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        let mut state = serializer.serialize_map(Some(self.len()))?;
        for (k, v) in self {
            state.serialize_entry(k, v)?;
        }
        state.end()
    }
}
```

Внедрить Deserialize для пользовательского типа карты

```
// A Visitor is a type that holds methods that a Deserializer can drive
// depending on what is contained in the input data.
//
// In the case of a map we need generic type parameters K and V to be
// able to set the output type correctly, but don't require any state.
// This is an example of a "zero sized type" in Rust. The PhantomData
// keeps the compiler from complaining about unused generic type
// parameters.
struct MyMapVisitor<K, V> {
    marker: PhantomData<MyMap<K, V>>
}

impl<K, V> MyMapVisitor<K, V> {
    fn new() -> Self {
        MyMapVisitor {
            marker: PhantomData
        }
    }
}

// This is the trait that Deserializers are going to be driving. There
// is one method for each type of data that our type knows how to
// deserialize from. There are many other methods that are not
// implemented here, for example deserializing from integers or strings.
// By default those methods will return an error, which makes sense
// because we cannot deserialize a MyMap from an integer or string.
impl<K, V> de::Visitor for MyMapVisitor<K, V>
    where K: Deserialize,
          V: Deserialize
{
    // The type that our Visitor is going to produce.
    type Value = MyMap<K, V>;

    // Deserialize MyMap from an abstract "map" provided by the
    // Deserializer. The MapVisitor input is a callback provided by
    // the Deserializer to let us see each entry in the map.
```

```
    fn visit_map<M>(self, mut visitor: M) -> Result<Self::Value, M::Error>
        where M: de::MapVisitor
    {
        let mut values = MyMap::with_capacity(visitor.size_hint().0);

        // While there are entries remaining in the input, add them
        // into our map.
        while let Some((key, value)) = visitor.visit()? {
            values.insert(key, value);
        }

        Ok(values)
    }

    // As a convenience, provide a way to deserialize MyMap from
    // the abstract "unit" type. This corresponds to `null` in JSON.
    // If your JSON contains `null` for a field that is supposed to
    // be a MyMap, we interpret that as an empty map.
    fn visit_unit<E>(self) -> Result<Self::Value, E>
        where E: de::Error
    {
        Ok(MyMap::new())
    }

    // When an unexpected data type is encountered, this method will
    // be invoked to inform the user what is actually expected.
    fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        write!(formatter, "a map or `null`")
    }
}

// This is the trait that informs Serde how to deserialize MyMap.
impl<K, V> Deserialize for MyMap<K, V>
    where K: Deserialize,
          V: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Instantiate our Visitor and ask the Deserializer to drive
        // it over the input data, resulting in an instance of MyMap.
        deserializer.deserialize_map(MyMapVisitor::new())
    }
}
```

Обработать массив значений без буферизации их в Vec

Предположим, у нас есть массив целых чисел, и мы хотим определить максимальное значение, не задерживая весь массив в памяти сразу. Этот подход может быть адаптирован для обработки множества других ситуаций, в которых данные должны обрабатываться при десериализации вместо них.

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;
use serde::{de, Deserialize, Deserializer};
```

```
use std::cmp;
use std::fmt;
use std::marker::PhantomData;

#[derive(Deserialize)]
struct Outer {
    id: String,

    // Deserialize this field by computing the maximum value of a sequence
    // (JSON array) of values.
    #[serde(deserialize_with = "deserialize_max")]
    // Despite the struct field being named `max_value`, it is going to come
    // from a JSON field called `values`.
    #[serde(rename(deserialize = "values"))]
    max_value: u64,
}

/// Deserialize the maximum of a sequence of values. The entire sequence
/// is not buffered into memory as it would be if we deserialize to Vec<T>
/// and then compute the maximum later.
///
/// This function is generic over T which can be any type that implements
/// Ord. Above, it is used with T=u64.
fn deserialize_max<T, D>(deserializer: D) -> Result<T, D::Error>
    where T: Deserialize + Ord,
          D: Deserializer
{
    struct MaxVisitor<T>(PhantomData<T>);

    impl<T> de::Visitor for MaxVisitor<T>
        where T: Deserialize + Ord
    {
        /// Return type of this visitor. This visitor computes the max of a
        /// sequence of values of type T, so the type of the maximum is T.
        type Value = T;

        fn expecting(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
            write!(formatter, "a sequence of numbers")
        }

        fn visit_seq<V>(self, mut visitor: V) -> Result<T, V::Error>
            where V: de::SeqVisitor
        {
            // Start with max equal to the first value in the seq.
            let mut max = match visitor.visit()? {
                Some(value) => value,
                None => {
                    // Cannot take the maximum of an empty seq.
                    let msg = "no values in seq when looking for maximum";
                    return Err(de::Error::custom(msg));
                }
            };

            // Update the max while there are additional values.
            while let Some(value) = visitor.visit()? {
                max = cmp::max(max, value);
            }

            Ok(max)
        }
    }
```

```
    // Create the visitor and ask the deserializer to drive it. The
    // deserializer will call visitor.visit_seq if a seq is present in
    // the input data.
    let visitor = MaxVisitor(PhantomData);
    deserializer.deserialize_seq(visitor)
}

fn main() {
    let j = r#"
        {
          "id": "demo-deserialize-max",
          "values": [
            256,
            100,
            384,
            314,
            271
          ]
        }
    "#;

    let out: Outer = serde_json::from_str(j).unwrap();

    // Prints "max value: 384"
    println!("max value: {}", out.max_value);
}
```

Рукописные ограничения общего типа

При выводе реализаций `Serialize` и `Deserialize` для структур с `Deserialize` параметрами типа, Serde может вывести правильные границы признаков без помощи программиста. Он использует несколько эвристик, чтобы угадать правильную границу, но, самое главное, она связывает `T: Serialize` с каждым параметром `T` который является частью сериализованного поля и границей `T: Deserialize` для каждого параметра `T` типа, который является частью десериализованное поле. Как и в случае с большинством эвристик, это не всегда правильно, и Серд обеспечивает выходной люк, чтобы заменить автоматически сгенерированную привязку на одну написанную программистом.

```
extern crate serde;
extern crate serde_json;
#[macro_use] extern crate serde_derive;

use serde::de::{self, Deserialize, Deserializer};

use std::fmt::Display;
use std::str::FromStr;

#[derive(Deserialize, Debug)]
struct Outer<'a, S, T: 'a + ?Sized> {
    // When deriving the Deserialize impl, Serde would want to generate a bound
    // `S: Deserialize` on the type of this field. But we are going to use the
    // type's `FromStr` impl instead of its `Deserialize` impl by going through
    // `deserialize_from_str`, so we override the automatically generated bound
    // by the one required for `deserialize_from_str`.
    #[serde(deserialize_with = "deserialize_from_str")]
```

```
    #[serde(bound(deserialize = "S: FromStr, S::Err: Display"))]
    s: S,

    // Here Serde would want to generate a bound `T: Deserialize`. That is a
    // stricter condition than is necessary. In fact, the `main` function below
    // uses T=str which does not implement Deserialize. We override the
    // automatically generated bound by a looser one.
    #[serde(bound(deserialize = "Ptr<'a, T>: Deserialize"))]
    ptr: Ptr<'a, T>,
}

/// Deserialize a type `S` by deserializing a string, then using the `FromStr`
/// impl of `S` to create the result. The generic type `S` is not required to
/// implement `Deserialize`.
fn deserialize_from_str<S, D>(deserializer: D) -> Result<S, D::Error>
    where S: FromStr,
          S::Err: Display,
          D: Deserializer
{
    let s: String = try!(Deserialize::deserialize(deserializer));
    S::from_str(&s).map_err(|e| de::Error::custom(e.to_string()))
}

/// A pointer to `T` which may or may not own the data. When deserializing we
/// always want to produce owned data.
#[derive(Debug)]
enum Ptr<'a, T: 'a + ?Sized> {
    Ref(&'a T),
    Owned(Box<T>),
}

impl<'a, T: 'a + ?Sized> Deserialize for Ptr<'a, T>
    where Box<T>: Deserialize
{
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        let box_t = try!(Deserialize::deserialize(deserializer));
        Ok(Ptr::Owned(box_t))
    }
}

fn main() {
    let j = r#"
        {
          "s": "1234567890",
          "ptr": "owned"
        }
    "#;

    let result: Outer<u64, str> = serde_json::from_str(j).unwrap();

    // result = Outer { s: 1234567890, ptr: Owned("owned") }
    println!("result = {:?}", result);
}
```

Внедрить сериализацию и десериализацию для типа в другом ящике

Правило согласования Rust требует, чтобы либо признак, либо тип, для которого вы

реализуете признак, должны быть определены в том же ящике, что и impl, поэтому невозможно реализовать Serialize и Deserialize для типа в другом ящике напрямую. Шаблон newtype и принуждение `Deref` обеспечивают способ реализации Serialize и Deserialize для типа, который ведет себя так же, как тот, который вы хотели.

```
use serde::{Serialize, Serializer, Deserialize, Deserializer};
use std::ops::Deref;

// Pretend this module is from some other crate.
mod not_my_crate {
    pub struct Data { /* ... */ }
}

// This single-element tuple struct is called a newtype struct.
struct Data(not_my_crate::Data);

impl Serialize for Data {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
        where S: Serializer
    {
        // Any implementation of Serialize.
    }
}

impl Deserialize for Data {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
        where D: Deserializer
    {
        // Any implementation of Deserialize.
    }
}

// Enable `Deref` coercion.
impl Deref for Data {
    type Target = not_my_crate::Data;
    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

// Now `Data` can be used in ways that require it to implement
// Serialize and Deserialize.
#[derive(Serialize, Deserialize)]
struct Outer {
    id: u64,
    name: String,
    data: Data,
}
```

Прочитайте Serde онлайн: https://riptutorial.com/ru/rust/topic/1170/serde

глава 7: Авто-разыменования

Examples

Оператор точки

. оператор в Rust поставляется с большим количеством магии! Когда вы используете . , компилятор будет вставлять столько * s (операции разыменования), необходимые для поиска метода по дереву дерева. Поскольку это происходит во время компиляции, не существует затрат времени на поиск метода.

```
let mut name: String = "hello world".to_string();
// no deref happens here because push is defined in String itself
name.push('!');

let name_ref: &String = &name;
// Auto deref happens here to get to the String. See below
let name_len = name_ref.len();
// You can think of this as syntactic sugar for the following line:
let name_len2 = (*name_ref).len();

// Because of how the deref rules work,
// you can have an arbitrary number of references.
// The . operator is clever enough to know what to do.
let name_len3 = (&&&&&&&&&&&&&name).len();
assert_eq!(name_len3, name_len);
```

Автоматическое разыменование также работает для любого типа, реализующего `std::ops::Deref` .

```
let vec = vec![1, 2, 3];
let iterator = vec.iter();
```

Здесь `iter` не является методом `Vec<T>` , а является методом `[T]` . Это работает , потому что `Vec<T>` реализует `Deref` с `Target=[T]` , который позволяет `Vec<T>` превращаются в `[T]` , когда разыменовываются по `*` оператора (который компилятор может вставить во время .).

Принудительные действия

Учитывая два типа `T` и `U` , `&T` будет принудительно (неявно преобразовывать) в `&U` тогда и только тогда, когда `T` реализует `Deref<Target=U>`

Это позволяет нам делать такие вещи:

```
fn foo(a: &[i32]) {
    // code
}
```

```
fn bar(s: &str) {
    // code
}

let v = vec![1, 2, 3];
foo(&v); // &Vec<i32> coerces into &[i32] because Vec<T> impls Deref<Target=[T]>

let s = "Hello world".to_string();
let rc = Rc::new(s);
// This works because Rc<T> impls Deref<Target=T> ∴ &Rc<String> coerces into
// &String which coerces into &str. This happens as much as needed at compile time.
bar(&rc);
```

Использование Deref и AsRef для аргументов функции

Для функций, которые необходимо взять набор объектов, срезы обычно являются хорошим выбором:

```
fn work_on_bytes(slice: &[u8]) {}
```

Поскольку `Vec<T>` и массивы `[T; N]` реализовать `Deref<Target=[T]>` , их можно легко принудить к фрагменту:

```
let vec = Vec::new();
work_on_bytes(&vec);

let arr = [0; 10];
work_on_bytes(&arr);

let slice = &[1,2,3];
work_on_bytes(slice); // Note lack of &, since it doesn't need coercing
```

Однако вместо того, чтобы явно требовать срез, можно сделать функцию для принятия любого типа, который *может* использоваться как срез:

```
fn work_on_bytes<T: AsRef<[u8]>>(input: T) {
    let slice = input.as_ref();
}
```

В этом примере функция `work_on_bytes` примет любой тип `T` который реализует `as_ref()` , который возвращает ссылку на `[u8]` .

```
work_on_bytes(vec);
work_on_bytes(arr);
work_on_bytes(slice);
work_on_bytes("strings work too!");
```

Реализация Deref для структуры Option и оболочки

```
use std::ops::Deref;
use std::fmt::Debug;
```

```
#[derive(Debug)]
struct RichOption<T>(Option<T>); // wrapper struct

impl<T> Deref for RichOption<T> {
    type Target = Option<T>; // Our wrapper struct will coerce into Option
    fn deref(&self) -> &Option<T> {
        &self.0 // We just extract the inner element
    }
}

impl<T: Debug> RichOption<T> {
    fn print_inner(&self) {
        println!("{:?}", self.0)
    }
}

fn main() {
    let x = RichOption(Some(1));
    println!("{:?}",x.map(|x| x + 1)); // Now we can use Option's methods...
    fn_that_takes_option(&x); // pass it to functions that take Option...
    x.print_inner() // and use it's own methods to extend Option
}

fn fn_that_takes_option<T : std::fmt::Debug>(x: &Option<T>) {
    println!("{:?}", x)
}
```

Пример Simple Deref

`Deref` есть простое правило: если у вас есть тип `T` и он реализует `Deref<Target=F>` , то `&T` коэрцирует в `&F` , компилятор будет повторять это столько раз, сколько необходимо для получения F, например:

```
fn f(x: &str) -> &str { x }
fn main() {
    // Compiler will coerce &&&&&&&str to &str and then pass it to our function
    f(&&&&&&&"It's a string");
}
```

Принудительное принуждение особенно полезно при работе с типами указателей, такими как `Box` или `Arc` , например:

```
fn main() {
    let val = Box::new(vec![1,2,3]);
    // Now, thanks to Deref, we still
    // can use our vector method as if there wasn't any Box
    val.iter().fold(0, |acc, &x| acc + x ); // 6
    // We pass our Box to the function that takes Vec,
    // Box<Vec> coerces to Vec
    f(&val)
}

fn f(x: &Vec<i32>) {
    println!("{:?}", x) // [1,2,3]
```

```
}
```

Прочитайте Авто-разыменования онлайн: https://riptutorial.com/ru/rust/topic/2574/авто-разыменования

глава 8: Аргументы командной строки

Вступление

Стандартная библиотека Rust не содержит правильный парсер аргументов (в отличие от `argparse` в Python), вместо этого предпочитая оставить это в сторонних ящиках. В этих примерах будет показано использование как стандартной библиотеки (для формирования грубого обработчика аргументов), так и библиотеки `clap` которая может более эффективно анализировать аргументы командной строки.

Синтаксис

- используйте std :: env; // Импортируем модуль env
- пусть args = env :: args (); // Храните итератор Args в переменной args.

Examples

Использование std :: env :: args ()

Вы можете получить доступ к аргументам командной строки, переданным вашей программе, используя функцию `std::env::args()` . Это возвращает итератор `Args` который вы можете перебрать или собрать в `Vec` .

Итерация через аргументы

```
use std::env;

fn main() {
    for argument in env::args() {
        if argument == "--help" {
            println!("You passed --help as one of the arguments!");
        }
    }
}
```

Сбор в `Vec`

```
use std::env;

fn main() {
    let arguments: Vec<String> = env::args().collect();
    println!("{} arguments passed", arguments.len());
}
```

Вы можете получить больше аргументов, чем вы ожидаете, если вы вызовете свою

программу следующим образом:

```
./example
```

Хотя похоже, что никакие аргументы не были переданы, первым аргументом является (**обычно**) имя исполняемого файла. Однако это не гарантия, поэтому вы всегда должны проверять и фильтровать аргументы, которые вы получаете.

Использование хлопка

Для более крупных программ командной строки использование `std::env::args()` довольно утомительно и трудно справляется. Вы можете использовать `clap` для обработки интерфейса командной строки, который будет анализировать аргументы, создавать справки и избегать ошибок.

Существует несколько *шаблонов,* которые вы можете использовать с `clap` , и каждый из них обеспечивает различную гибкость.

Шаблон Builder

Это самый подробный (и гибкий) метод, поэтому он полезен, когда вам нужен мелкомасштабный контроль над вашим CLI.

`clap` различает *подкоманды* и *аргументы* . Подкоманды действуют как независимые подпрограммы в вашей основной программе, точно так же, как `cargo run` и `git push` . Они могут иметь свои собственные параметры командной строки и входные данные. Аргументы - это простые флаги, такие как `--verbose` , и они могут принимать входные данные (например, - `--message "Hello, world"`).

```
extern crate clap;
use clap::{Arg, App, SubCommand};

fn main() {
    let app = App::new("Foo Server")
        .about("Serves foos to the world!")
        .version("v0.1.0")
        .author("Foo (@Example on GitHub)")
        .subcommand(SubCommand::with_name("run")
            .about("Runs the Foo Server")
            .arg(Arg::with_name("debug")
                .short("D")
                .about("Sends debug foos instead of normal foos.")))

    // This parses the command-line arguments for use.
    let matches = app.get_matches();

    // We can get the subcommand used with matches.subcommand(), which
    // returns a tuple of (&str, Option<ArgMatches>) where the &str
    // is the name of the subcommand, and the ArgMatches is an
    // ArgMatches struct:
    // https://docs.rs/clap/2.13.0/clap/struct.ArgMatches.html
```

```
    if let ("run", Some(run_matches)) = app.subcommand() {
        println!("Run was used!");
    }
}
```

Прочитайте Аргументы командной строки онлайн: https://riptutorial.com/ru/rust/topic/7015/аргументы-командной-строки

глава 9: вариант

Вступление

Тип `Option<T>` - это эквивалент NULL, равный Rust, без всех проблем, которые возникают с ним. Большинство C-подобных языков допускают, что любая переменная имеет значение `null` если нет данных, но тип `Option` вдохновлен функциональными языками, которые предпочитают «опциональные» (например, Hadell's `Maybe` monad). Использование `Option` типов позволит вам выразить мысль , что данные могут или не могут быть там (так как ржавчина не обнуляемые типов).

Examples

Создание значения параметра и соответствия шаблону

```
// The Option type can either contain Some value or None.
fn find(value: i32, slice: &[i32]) -> Option<usize> {
    for (index, &element) in slice.iter().enumerate() {
        if element == value {
            // Return a value (wrapped in Some).
            return Some(index);
        }
    }
    // Return no value.
    None
}

fn main() {
    let array = [1, 2, 3, 4, 5];
    // Pattern match against the Option value.
    if let Some(index) = find(2, &array) {
        // Here, there is a value.
        println!("The element 2 is at index {}.", index);
    }

    // Check if the result is None (no value).
    if let None = find(12, &array) {
        // Here, there is no value.
        println!("The element 12 is not in the array.");
    }

    // You can also use `is_some` and `is_none` helpers
    if find(12, &array).is_none() {
        println!("The element 12 is not in the array.");
    }
}
```

Уничтожение опциона

```
fn main() {
```

```
    let maybe_cake = Some("Chocolate cake");
    let not_cake = None;

    // The unwrap method retrieves the value from the Option
    // and panics if the value is None
    println!("{}", maybe_cake.unwrap());

    // The expect method works much like the unwrap method,
    // but panics with a custom, user provided message.
    println!("{}", not_cake.expect("The cake is a lie."));

    // The unwrap_or method can be used to provide a default value in case
    // the value contained within the option is None. This example would
    // print "Cheesecake".
    println!("{}", not_cake.unwrap_or("Cheesecake"));

    // The unwrap_or_else method works like the unwrap_or method,
    // but allows us to provide a function which will return the
    // fallback value. This example would print "Pumpkin Cake".
    println!("{}", not_cake.unwrap_or_else(|| { "Pumpkin Cake" }));

    // A match statement can be used to safely handle the possibility of none.
    match maybe_cake {
        Some(cake) => println!("{} was consumed.", cake),
        None       => println!("There was no cake.")
    }

    // The if let statement can also be used to destructure an Option.
    if let Some(cake) = maybe_cake {
        println!("{} was consumed.", cake);
    }
}
```

Развертывание ссылки на Опцию, владеющую ее содержимым

Ссылка на параметр `&Option<T>` не может быть развернута, если тип `T` не может быть скопирован. Решение заключается в изменении опции `&Option<&T>` с использованием `as_ref()` .

Ржавчина запрещает передачу права собственности на объекты при заимствовании объектов. Когда сам Опцион заимствован (`&Option<T>`), его содержимое также - косвенно - заимствовано.

```
#[derive(Debug)]
struct Foo;

fn main() {
    let wrapped = Some(Foo);
    let wrapped_ref = &wrapped;

    println!("{:?}", wrapped_ref.unwrap()); // Error!
}
```

не может выйти из заемного контента [--explain E0507]

Тем не менее, можно создать ссылку на содержимое `Option<T>` . Опция `as_ref()` метод

возвращает параметр для `&T` , который может быть развернут без передачи права собственности:

```
println!("{:?}", wrapped_ref.as_ref().unwrap());
```

Использование опции с картой и and_then

Операция `map` является полезным инструментом при работе с массивами и векторами, но также может использоваться для функционального использования значений `Option` .

```
fn main() {

    // We start with an Option value (Option<i32> in this case).
    let some_number = Some(9);

    // Let's do some consecutive calculations with our number.
    // The crucial point here is that we don't have to unwrap
    // the content of our Option type - instead, we're just
    // transforming its content. The result of the whole operation
    // will still be an Option<i32>. If the initial value of
    // 'some_number' was 'None' instead of 9, then the result
    //  would also be 'None'.
    let another_number = some_number
        .map(|n| n - 1) // => Some(8)
        .map(|n| n * n) // => Some(64)
        .and_then(|n| divide(n, 4)); // => Some(16)

    // In the last line above, we're doing a division using a helper
    // function (definition: see bottom).
    // 'and_then' is very similar to 'map', but allows us to pass a
    // function which returns an Option type itself. To ensure that we
    // don't end up with Option<Option<i32>>, 'and_then' flattens the
    // result (in other languages, 'and_then' is also known as 'flatmap').

    println!("{}", to_message(another_number));
    // => "16 is definitely a number!"

    // For the sake of completeness, let's check the result when
    // dividing by zero.
    let final_number = another_number
        .and_then(|n| divide(n, 0)); // => None

    println!("{}", to_message(final_number));
    // => "None!"
}

// Just a helper function for integer division. In case
// the divisor is zero, we'll get 'None' as result.
fn divide(number: i32, divisor: i32) -> Option<i32> {
    if divisor != 0  { Some(number/divisor) } else { None }
}

// Creates a message that tells us whether our
// Option<i32> contains a number or not. There are other
// ways to achieve the same result, but let's just use
// map again!
fn to_message(number: Option<i32>) -> String {
```

```
    number
        .map(|n| format!("{} is definitely a number!", n)) // => Some("...")
        .unwrap_or("None!".to_string()) // => "..."
}
```

Прочитайте вариант онлайн: https://riptutorial.com/ru/rust/topic/1125/вариант

глава 10: Владение

Вступление

Собственность - одна из самых важных концепций в Rust, и это то, чего нет на большинстве других языков. Идея о том, что ценность может *принадлежать* определенной переменной, часто довольно трудно понять, особенно в языках, где копирование является неявным, но в этом разделе будут рассмотрены различные идеи, связанные с владением.

Синтаксис

- пусть x: & T = ... // x - неизменяемая ссылка
- пусть x: & mut T = ... // x - исключительная, изменяемая ссылка
- пусть _ = & mut foo; // заимствовать foo изменчиво (т. е. исключительно)
- пусть _ = & foo; // заимствовать foo неизменно
- пусть _ = foo; // перемещение foo (требуется владение)

замечания

- В гораздо более ранних версиях Rust (до 1.0, май 2015 г.) у принадлежащей переменной был тип, начинающийся с ~ . Вы можете увидеть это в очень старых примерах.

Examples

Собственность и займы

Все значения в Rust имеют ровно один владелец. Владелец несет ответственность за удаление этого значения, когда он выходит за пределы области действия, и является единственным, кто может *переместить* право собственности на это значение. Владелец значения может отдать *ссылки* на него, разрешив другим частям кода *занять* это значение. В любой момент времени может быть любое количество неизменяемых ссылок на значение:

```
let owned = String::from("hello");
// since we own the value, we may let other variables borrow it
let immutable_borrow1 = &owned;
// as all current borrows are immutable, we can allow many of them
let immutable_borrow2 = &owned;
// in fact, if we have an immutable reference, we are also free to
// duplicate that reference, since we maintain the invariant that
// there are only immutable references
let immutable_borrow3 = &*immutable_borrow2;
```

или одну изменяемую ссылку на нее (`ERROR` обозначает ошибку времени компиляции):

```
// for us to borrow a value mutably, it must be mutable
let mut owned = String::from("hello");
// we can borrow owned mutably
let mutable_borrow = &mut owned;
// but note that we cannot borrow owned *again*
let mutable_borrow2 = &mut owned; // ERROR, already borrowed
// nor can we cannot borrow owned immutably
// since a mutable borrow is exclusive.
let immutable_borrow = &owned; // ERROR, already borrowed
```

Если к значению имеются невыполненные ссылки (изменяемые или неизменяемые), это значение не может быть перемещено (т. Е. Его собственность отдана). Мы должны были бы убедиться, что все ссылки были отброшены первым, чтобы разрешить перемещать значение:

```
let foo = owned; // ERROR, outstanding references to owned
let owned = String::from("hello");
{
    let borrow = &owned;
    // ...
} // the scope ends the borrow
let foo = owned; // OK, owned and not borrowed
```

Периодичность и продолжительность жизни

Все значения в Rust имеют *всю жизнь* . Время жизни значения охватывает сегмент кода от значения, вводимого туда, где он перемещен, или конец области сложения

```
{
    let x = String::from("hello"); //             +
    // ...                                       :
    let y = String::from("hello"); //      +      |
    // ...                                 :      :
    foo(x) // x is moved                   |      = x's lifetime
    // ...                                 :
} //                                       = y's lifetime
```

Всякий раз, когда вы заимствуете значение, результирующая ссылка имеет *срок службы* , привязанный к времени жизни заимствованного значения:

```
{
    let x = String::from("hello");
    let y = String::from("world");
    // when we borrow y here, the lifetime of the reference
    // stored in foo is equal to the lifetime of y
    // (i.e., between let y = above, to the end of the scope below)
    let foo = &y;
    // similarly, this reference to x is bound to the lifetime
    // of x --- bar cannot, for example, spawn a thread that uses
    // the reference beyond where x is moved below.
    bar(&x);
```

```
}
```

Звонки и вызов функций

Большинство вопросов о собственности возникают при написании функций. Когда вы указываете типы аргументов функции, вы можете выбрать *способ* передачи этого значения. Если вам нужен только доступ только для чтения, вы можете взять неизменяемую ссылку:

```
fn foo(x: &String) {
    // foo is only authorized to read x's contents, and to create
    // additional immutable references to it if it so desires.
    let y = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // ERROR, cannot mutate with immutable reference
    println!("{}", x.len()); // reading OK
    foo(x); // forwarding reference OK
}
```

Если `foo` необходимо изменить аргумент, он должен взять исключительную, изменяемую ссылку:

```
fn foo(x: &mut String) {
    // foo is still not responsible for dropping x before returning,
    // nor is it allowed to. however, foo may modify the String.
    let x2 = *x; // ERROR, cannot move when not owned
    x.push_str("foo"); // mutating OK
    drop(*x); // ERROR, cannot drop value when not owned
    println!("{}", x.len()); // reading OK
}
```

Если вы не укажете ни `&` или `&mut` , вы говорите, что функция будет владеть аргументом. Это означает, что `foo` теперь также несет ответственность за удаление `x` .

```
fn foo(x: String) {
    // foo may do whatever it wishes with x, since no-one else has
    // access to it. once the function terminates, x will be dropped,
    // unless it is moved away when calling another function.
    let mut x2 = x; // moving OK
    x2.push_str("foo"); // mutating OK
    let _ = &mut x2; // mutable borrow OK
    let _ = &x2; // immutable borrow OK (note that &mut above is dropped)
    println!("{}", x2.len()); // reading OK
    drop(x2); // dropping OK
}
```

Собственность и свойство копирования

Некоторые типы Rust реализуют свойство `Copy` . Типы, которые являются `Copy` могут быть перемещены без использования соответствующего значения. Это связано с тем, что содержимое значения можно просто скопировать по байтам в память для получения нового идентичного значения. Большинство примитивов в Rust (`bool` , `usize` , `f64` и т. Д.) -

это `Copy` .

```
let x: isize = 42;
let xr = &x;
let y = *xr; // OK, because isize is Copy
// both x and y are owned here
```

Примечательно, что `Vec` и `String` *не* `Copy` :

```
let x = Vec::new();
let xr = &x;
let y = *xr; // ERROR, cannot move out of borrowed content
```

Прочитайте Владение онлайн: https://riptutorial.com/ru/rust/topic/4395/владение

глава 11: Время жизни

Синтаксис

- fn функция <'a> (x: &' a Тип)
- struct Struct <'a> {x: &' a Тип}
- enum Enum <'a> {Вариант (&' a Тип)}
- impl <'a> Struct <' a> {fn x <'a> (& self) -> &' a Type {self.x}}
- impl <'a> Значение <' a> для типа
- impl <'a> Trait для Type <' a>
- `fn function<F>(f: F) where for<'a> F: FnOnce(&'a Type)`
- `struct Struct<F> where for<'a> F: FnOnce(&'a Type) { x: F }`
- `enum Enum<F> where for<'a> F: FnOnce(&'a Type) { Variant(F) }`
- `impl<F> Struct<F> where for<'a> F: FnOnce(&'a Type) { fn x(&self) -> &F { &self.x } }`

замечания

- Все ссылки в Rust имеют всю жизнь, даже если они явно не аннотируются. Компилятор способен неявно назначать сроки жизни.
- `'static` время жизни назначается для ссылок, которые хранятся в двоичном коде программы и будут действительны на протяжении всего его выполнения. Это время жизни больше всего относится к строковым литералам, которые имеют тип `&'static str` .

Examples

Параметры функции (время ввода)

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

Это указывает на то, что `foo` имеет время жизни `'a` , а параметр `x` должен иметь время жизни не менее `'a` . Функциональные времена жизни обычно пропускаются по времени *жизни* :

```
fn foo(x: &u32) {
    // ...
}
```

В случае, когда функция принимает несколько ссылок в качестве параметров и возвращает ссылку, компилятор не может вывести время жизни результата через *пожизненный elision* .

```
error[E0106]: missing lifetime specifier
1 | fn foo(bar: &str, baz: &str) -> &i32 {
  |                                 ^ expected lifetime parameter
```

Вместо этого параметры срока жизни должны быть явно указаны.

```
// Return value of `foo` is valid as long as `bar` and `baz` are alive.
fn foo<'a>(bar: &'a str, baz: &'a str) -> &'a i32 {
```

Функции также могут принимать несколько параметров времени жизни.

```
// Return value is valid for the scope of `bar`
fn foo<'a, 'b>(bar: &'a str, baz: &'b str) -> &'a i32 {
```

Поля структуры

```
struct Struct<'a> {
    x: &'a u32,
}
```

Это указывает, что любой данный экземпляр `Struct` имеет время жизни `'a` , а `&u32` хранящийся в `x` должен иметь время жизни не менее `'a` .

Блокировка Impl

```
impl<'a> Type<'a> {
    fn my_function(&self) -> &'a u32 {
        self.x
    }
}
```

Это указывает, что `Type` имеет время жизни `'a` и что ссылка, возвращаемая `my_function()` может быть более недействительной после того, как `'a` заканчивается, потому что `Type` больше не существует для сохранения `self.x`

Границы уровня более высокого ранга

```
fn copy_if<F>(slice: &[i32], pred: F) -> Vec<i32>
    where for<'a> F: Fn(&'a i32) -> bool
{
    let mut result = vec![];
    for &element in slice {
        if pred(&element) {
            result.push(element);
        }
    }
    result
}
```

Это указывает, что ссылка на i32 в привязке к значению `Fn` может иметь любое время жизни.

Не работает следующее:

```
fn wrong_copy_if<'a, F>(slice: &[i32], pred: F) -> Vec<i32>
    where F: Fn(&'a i32) -> bool
{                                   // <-----------------+
    let mut result = vec![];        //       'a scope   |
    for &element in slice {         // <--------+       |
        if pred(&element) {         //          |       |
            result.push(element);   // element's|       |
        }                           //   scope  |       |
    }                               // <--------+       |
    result                          //                  |
}                                   // <-----------------+
```

Компилятор дает следующую ошибку:

```
error: `element` does not live long enough
if pred(&element) {         //          |       |
         ^~~~~~~
```

потому что локальная переменная `element` не живет так долго, как `'a` жизни» (как мы видим из комментариев кода).

Время жизни не может быть объявлено на уровне функции, потому что нам нужно другое время жизни. Вот почему мы использовали `for<'a>` : чтобы указать, что ссылка может быть действительной для любого времени жизни (следовательно, можно использовать меньшее время жизни).

Оценки границ более высокого ранга также могут быть использованы для структур:

```
struct Window<F>
    where for<'a> F: FnOnce(&'a Window<F>)
{
    on_close: F,
}
```

а также на другие предметы.

Оценки признаков более высокого ранга наиболее часто используются с чертами `Fn*` .

Для этих примеров время жизни elision работает нормально, поэтому нам не нужно указывать сроки жизни.

Прочитайте Время жизни онлайн: https://riptutorial.com/ru/rust/topic/2074/время-жизни

глава 12: Вставные значения

Вступление

Коробки - очень важная часть ржавчины, и каждый расторопник должен знать, что это такое и как их использовать

Examples

Создание коробки

В стабильном Rust вы создаете Box, используя функцию `Box::new` .

```
let boxed_int: Box<i32> = Box::new(1);
```

Использование значений в штучной упаковке

Поскольку Boxes реализуют `Deref<Target=T>` , вы можете использовать значения в `Deref<Target=T>` как и значение, которое они содержат.

```
let boxed_vec = Box::new(vec![1, 2, 3]);
println!("{}", boxed_vec.get(0));
```

Если вы хотите сопоставить шаблон по размеру в коробке, вам может потребоваться разыменовать поле вручную.

```
struct Point {
    x: i32,
    y: i32,
}

let boxed_point = Box::new(Point { x: 0, y: 0});
// Notice the *. That dereferences the boxed value into just the value
match *boxed_point {
    Point {x, y} => println!("Point is at ({}, {})", x, y),
}
```

Использование ящиков для создания рекурсивных перечислений и структур

Если вы попытаетесь создать рекурсивное перечисление в Rust без использования Box, вы получите ошибку времени компиляции, указав, что перечисление не может быть задано.

```
// This gives an error!
enum List {
```

```
    Nil,
    Cons(i32, List)
}
```

Чтобы перечисление имело определенный размер, рекурсивно содержащее значение должно быть в поле.

```
// This works!
enum List {
    Nil,
    Cons(i32, Box<List>)
}
```

Это работает, потому что Box всегда имеет тот же размер независимо от того, что такое T, что позволяет Rust давать список размером.

Прочитайте Вставные значения онлайн: https://riptutorial.com/ru/rust/topic/9341/вставные-значения

глава 13: Встроенная сборка

Синтаксис

- #! [feature (asm)] // Включить asm! встроенный макрос
- asm! (<template>: <output>: <input>: <clobbers>: <options>) // Испускаем шаблон сборки (например, «NOP», «ADD% eax, 4») с заданными параметрами.

Examples

Асм! макрос

Встроенная сборка будет поддерживаться только в ночных версиях Rust до стабилизации . Чтобы включить использование `asm!` макроса, используйте следующий атрибут атрибута в верхней части основного файла (*затвор функции*):

```
#![feature(asm)]
```

Затем используйте `asm!` макрос в любом `unsafe` блоке:

```
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }

    // asm!("NOP");
    // That would be invalid here, because we are no longer in an
    // unsafe block.
}
```

Условно компилировать встроенную сборку

Используйте условную компиляцию, чтобы обеспечить компиляцию кода только для предполагаемого набора команд (например, `x86`). В противном случае код может стать недействительным, если программа скомпилирована для другой архитектуры, такой как ARM-процессоры.

```
#![feature(asm)]

// Any valid x86 code is valid for x86_64 as well. Be careful
// not to write x86_64 only code while including x86 in the
// compilation targets!
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn do_nothing() {
    unsafe {
        asm!("NOP");
    }
```

```
}

#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn do_nothing() {
    // This is an alternative implementation that doesn't use any asm!
    // calls. Therefore, it should be safe to use as a fallback.
}
```

Входы и выходы

```
#![feature(asm)]

#[cfg(any(target_arch="x86", target_arch="x86_64"))]
fn subtract(first: i32, second: i32) {
   unsafe {
        // Output values must either be unassigned (let result;) or mutable.
        let result: i32;
        // Each value that you pass in will be in a certain register, which
        // can be accessed with $0, $1, $2...
        //
        // The registers are assigned from left to right, so $0 is the
        // register containing 'result', $1 is the register containing
        // 'first' and $2 is the register containing 'second'.
        //
        // Rust uses AT&T syntax by default, so the format is:
        // SUB source, destination
        // which is equivalent to:
        // destination -= source;
        //
        // Because we want to subtract the first from the second,
        // we use the 0 constraint on 'first' to use the same
        // register as the output.
        // Therefore, we're doing:
        // SUB second, first
        // and getting the value of 'first'

        asm!("SUB $2, $0" : "=r"(result) : "0"(first), "r"(second));
        println!("{}", result);
    }
}
```

Коды ограничений LLVM можно найти здесь , но это может различаться в зависимости от версии LLVM, используемой вашим компилятором `rustc` .

Прочитайте Встроенная сборка онлайн: https://riptutorial.com/ru/rust/topic/6998/встроенная-сборка

глава 14: Генерация случайных чисел

Вступление

Rust имеет встроенную возможность обеспечения генерации случайных чисел через ящик *rand* . Будучи частью стандартной библиотеки Rust, функциональность ящика *rand* была разделена, чтобы ее развитие стабилизировалось отдельно от остальной части проекта Rust. В этом разделе рассказывается, как просто добавить ящик rand, затем генерировать и выводить случайное число в Rust.

замечания

Существует встроенная поддержка RNG, связанная с каждым потоком, хранящимся в локальном хранилище потоков. К этому RNG можно получить доступ через `thread_rng` или использовать неявно через `random` . Этот RNG обычно случайным образом высеивается из источника случайной системы операционной системы, например `/dev/urandom` в Unix-системах, и автоматически генерируется из этого источника после генерации 32 KiB случайных данных.

Приложение, для которого требуется источник энтропии для криптографических целей, должно использовать `OsRng` , который считывает случайность из источника, который предоставляет операционная система (например, `/dev/urandom` в Unixes или `CryptGenRandom()` в Windows). Другие генераторы случайных чисел, предоставленные этим модулем, не подходят для таких целей.

Examples

Создание двух случайных чисел с Rand

Во-первых, вам нужно добавить ящик в файл Cargo.toml в качестве зависимости.

```
[dependencies]
rand = "0.3"
```

Это доставит ящик `rand` из ящиков.io . Затем добавьте это в свой корневой каталог.

```
extern crate rand;
```

Поскольку этот пример будет обеспечивать простой вывод через терминал, мы создадим основную функцию и напечатаем два случайно генерируемых номера на консоли. В этом примере будет создан кешированный локальный генератор случайных чисел. При создании нескольких значений это часто оказывается более эффективным.

```
use rand::Rng;

fn main() {

    let mut rng = rand::thread_rng();

    if rng.gen() { // random bool
        println!("i32: {}, u32: {}", rng.gen::<i32>(), rng.gen::<u32>())
    }
}
```

Когда вы запускаете этот пример, вы должны увидеть следующий ответ в консоли.

```
$ cargo run
     Running `target/debug/so`
i32: 1568599182, u32: 2222135793
```

Создание персонажей с Rand

Чтобы генерировать символы, вы можете использовать функцию генератора случайных чисел в потоке, `random` .

```
fn main() {
    let tuple = rand::random::<(f64, char)>();
    println!("{:?}", tuple)
}
```

Для случайных или сингулярных запросов, например выше, это разумный эффективный метод. Однако, если вы собираетесь генерировать больше, чем несколько чисел, вы обнаружите, что кеширование генератора будет более эффективным.

В этом случае вы должны ожидать увидеть следующий результат.

```
$ cargo run
     Running `target/debug/so`
(0.906881, '\u{9edc}')
```

Прочитайте Генерация случайных чисел онлайн: https://riptutorial.com/ru/rust/topic/8864/генерация-случайных-чисел

глава 15: Глобалы

Синтаксис

- const IDENTIFIER: type = constexpr;
- static [mut] IDENTIFIER: type = expr;
- lazy_static! {static ref IDENTIFIER: type = expr; }

замечания

- `const` значения всегда встраиваемые и не имеют адресов в памяти.
- `static` значения никогда не привязаны и имеют один экземпляр с фиксированным адресом.
- `static mut` values не являются безопасными для памяти и поэтому могут быть доступны только в `unsafe` блоке.
- Иногда использование глобальных статических изменяемых переменных в многопоточном коде может быть опасным, поэтому рассмотрите использование std :: sync :: Mutex или других альтернатив
- Объекты `lazy_static` неизменяемы, инициализируются только один раз, распределяются между всеми потоками и могут быть напрямую доступны (нет связанных типов обертки). Напротив, объекты `thread_local` предназначены для изменения, инициализируются один раз для каждого потока, а обращения являются косвенными (с использованием типа оболочки `LocalKey<T>`)

Examples

Const

Ключевое слово `const` объявляет глобальную привязку константы.

```
const DEADBEEF: u64 = 0xDEADBEEF;

fn main() {
    println("{:X}", DEADBEEF);
}
```

Эти результаты

```
DEADBEEF
```

статический

`static` ключевое слово объявляет глобальную статическую привязку, которая может быть

изменчивой.

```
static HELLO_WORLD: &'static str = "Hello, world!";

fn main() {
    println("{}", HELLO_WORLD);
}
```

Эти результаты

```
Hello, world!
```

lazy_static!

Используйте ящик `lazy_static` для создания глобальных неизменяемых переменных, которые инициализируются во время выполнения. Мы используем `HashMap` в качестве демонстрации.

В `Cargo.toml` :

```
[dependencies]
lazy_static = "0.1.*"
```

В `main.rs` :

```
#[macro_use]
extern crate lazy_static;

lazy_static! {
    static ref HASHMAP: HashMap<u32, &'static str> = {
        let mut m = HashMap::new();
        m.insert(0, "hello");
        m.insert(1, ",");
        m.insert(2, " ");
        m.insert(3, "world");
        m
    };
    static ref COUNT: usize = HASHMAP.len();
}

fn main() {
    // We dereference COUNT because it's type is &usize
    println!("The map has {} entries.", *COUNT);

    // Here we don't dereference with * because of Deref coercions
    println!("The entry for `0` is \"{}\".", HASHMAP.get(&0).unwrap());
}
```

Потоковые локальные объекты

Локальный объект потока инициализируется при первом использовании в потоке. И, как следует из названия, каждый поток будет получать новую копию, независимую от других

потоков.

```
use std::cell::RefCell;
use std::thread;

thread_local! {
    static FOO: RefCell<f32> = RefCell::new(1.0);
}

// When this macro expands, `FOO` gets type `thread::LocalKey<RefCell<f32>>`.
//
// Side note: One of its private member is a pointer to a function which is
// responsible for returning the thread-local object. Having all its members
// `Sync` [0], `LocalKey` is also implicitly `Sync`.
//
// [0]: As of writing this, `LocalKey` just has 2 function-pointers as members

fn main() {
    FOO.with(|foo| {
        // `foo` is of type `&RefCell<f64>`
        *foo.borrow_mut() = 3.0;
    });

    thread::spawn(move|| {
        // Note that static objects do not move (`FOO` is the same everywhere),
        // but the `foo` you get inside the closure will of course be different.
        FOO.with(|foo| {
            println!("inner: {}", *foo.borrow());
        });
    }).join().unwrap();

    FOO.with(|foo| {
        println!("main: {}", *foo.borrow());
    });
}
```

Выходы:

```
inner: 1
main: 3
```

Безопасный статический мут с mut_static

Смещаемые глобальные элементы (называемые `static mut` , выделяющие неотъемлемое противоречие, связанное с их использованием) являются небезопасными, поскольку компилятору сложно обеспечить их надлежащее использование.

Тем не менее, введение взаимоисключающих блокировок вокруг данных позволяет безопасные для памяти изменяемые глобальные переменные. Это НЕ означает, что они логически безопасны!

```
#[macro_use]
extern crate lazy_static;
extern crate mut_static;
```

```
use mut_static::MutStatic;

pub struct MyStruct { value: usize }

impl MyStruct {
    pub fn new(v: usize) -> Self{
        MyStruct { value: v }
    }
    pub fn getvalue(&self) -> usize { self.value }
    pub fn setvalue(&mut self, v: usize) { self.value = v }
}

lazy_static! {
    static ref MY_GLOBAL_STATE: MutStatic<MyStruct> = MutStatic::new();
}

fn main() {
    // Here, I call .set on the MutStatic to put data inside it.
    // This can fail.
    MY_GLOBAL_STATE.set(MyStruct::new(0)).unwrap();
    {
        // Using the global state immutably is easy...
        println!("Before mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
         // Using it mutably is too...
         let mut mut_handle = MY_GLOBAL_STATE.write().unwrap();
         mut_handle.setvalue(3);
         println!("Changed value to 3.");
    }
    {
        // As long as there's a scope change we can get the
        // immutable version again...
        println!("After mut: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }
    {
        // But beware! Anything can change global state!
        foo();
        println!("After foo: {}",
                 MY_GLOBAL_STATE.read().unwrap().getvalue());
    }

}

// Note that foo takes no parameters
fn foo() {
    let val;
    {
        val = MY_GLOBAL_STATE.read().unwrap().getvalue();
    }
    {
        let mut mut_handle =
            MY_GLOBAL_STATE.write().unwrap();
        mut_handle.setvalue(val + 1);
    }
}
```

Этот код выводит результат:

```
Before mut: 0
Changed value to 3.
After mut: 3
After foo: 4
```

Это не то, что должно произойти в Rust. `foo()` не принимала изменчивую ссылку на что-либо, поэтому она не должна была ничего мутировать, и все же она сделала это. Это может привести к очень трудным для отладки логических ошибок.

С другой стороны, это иногда то, что вы хотите. Например, для многих игровых движков требуется глобальный кэш изображений и других ресурсов, которые лениво загружаются (или используют какую-то другую сложную стратегию загрузки). MutStatic идеально подходит для этой цели.

Прочитайте Глобалы онлайн: https://riptutorial.com/ru/rust/topic/1244/глобалы

глава 16: Голая металлическая ржавчина

Вступление

Стандартная библиотека Rust (`std`) компилируется только для нескольких архитектур. Итак, чтобы скомпилировать другие архитектуры (которые поддерживает LLVM), программы Rust могут отказаться от использования всего `std` и вместо этого использовать только переносное подмножество, известное как The Core Library (`core`).

Examples

#! [no_std] Привет, Мир!

```
#![feature(start, libc, lang_items)]
#![no_std]
#![no_main]

// The libc crate allows importing functions from C.
extern crate libc;

// A list of C functions that are being imported
extern {
    pub fn printf(format: *const u8, ...) -> i32;
}

#[no_mangle]
// The main function, with its input arguments ignored, and an exit status is returned
pub extern fn main(_nargs: i32, _args: *const *const u8) -> i32 {
    // Print "Hello, World" to stdout using printf
    unsafe {
        printf(b"Hello, World!\n" as *const u8);
    }

    // Exit with a return status of 0.
    0
}

#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] extern fn panic_fmt() -> ! { panic!() }
```

Прочитайте Голая металлическая ржавчина онлайн: https://riptutorial.com/ru/rust/topic/8344/голая-металлическая-ржавчина

глава 17: грузовой

Вступление

Cargo - менеджер пакетов Rust, используемый для управления *ящиками* (термин Rust для библиотек / пакетов). Cargo преимущественно извлекает пакеты из crates.io и может управлять сложными деревьями зависимостей с определенными требованиями к версии (с использованием семантического управления версиями). Cargo также может помочь в строительстве, эксплуатации и управлении проектами Rust с `cargo build` , `cargo run` и `cargo test` (среди других полезных команд).

Синтаксис

- груз new crate_name [--bin]
- груз init [--bin]
- сборка груза [--release]
- грузовой ход [--release]
- проверка груза
- испытание груза
- грузовой стенд
- обновление груза
- грузовой пакет
- опубликовать груз
- [un] установить binary_crate_name
- поиск груза crate_name
- грузовая версия
- Вход для грузовика api_key

замечания

- В настоящий момент подкоманда `cargo bench` требует, чтобы ночная версия компилятора работала эффективно.

Examples

Создать новый проект

Библиотека

```
cargo new my-library
```

Это создает новый каталог с именем `my-library` содержащий файл конфигурации груза и исходный каталог, содержащий единственный исходный файл Rust:

```
my-library/Cargo.toml
my-library/src/lib.rs
```

Эти два файла уже содержат базовый скелет библиотеки, так что вы можете сразу же пройти `cargo test` на `cargo test` (из каталога `my-library`), чтобы проверить, все ли работает.

двоичный

```
cargo new my-binary --bin
```

Это создает новый каталог с именем `my-binary` с аналогичной структурой, такой как библиотека:

```
my-binary/Cargo.toml
my-binary/src/main.rs
```

На этот раз `cargo` установит простой Hello World, который мы сможем запустить сразу с `cargo run` .

Вы также можете создать новый проект в текущем каталоге с помощью подкатегории `init` :

```
cargo init --bin
```

Как и выше, удалите флаг `--bin` для создания нового проекта библиотеки. Имя текущей папки автоматически используется как имя корзины.

Проект строительства

отлаживать

```
cargo build
```

Релиз

Создание с `--release` флага `--release` допускает определенные оптимизации компилятора,

которые не выполняются при построении отладочной сборки. Это заставляет код работать быстрее, но время компиляции еще больше. Для обеспечения оптимальной производительности эта команда должна использоваться после того, как сборка готово.

```
cargo build --release
```

Тестирование

Основное использование

```
cargo test
```

Показать выход программы

```
cargo test -- --nocapture
```

Запустить конкретный пример

```
cargo test test_name
```

Привет, мирская программа

Это сеанс оболочки, показывающий, как создать программу «Hello world» и запустить ее с помощью Cargo:

```
$ cargo new hello --bin
$ cd hello
$ cargo run
   Compiling hello v0.1.0 (file:///home/rust/hello)
     Running `target/debug/hello`
Hello, world!
```

После этого вы можете редактировать программу, открыв `src/main.rs` в текстовом редакторе.

Публикация ящика

Чтобы опубликовать ящик на crates.io , вы должны войти в систему с Cargo (см. « *Подключение груза к учетной записи Crates.io* »).

Вы можете упаковать и опубликовать свой ящик со следующими командами:

```
cargo package
cargo publish
```

Любые ошибки в вашем файле `Cargo.toml` будут выделены во время этого процесса. Вы должны убедиться, что **обновите свою версию** и убедитесь, что ваш `.gitignore` или `Cargo.toml` исключает любые нежелательные файлы.

Подключение груза к учетной записи Crates.io

Учетные записи crates.io создаются путем входа в систему с GitHub; вы не можете подписаться на какой-либо другой метод.

Чтобы подключить свою учетную запись GitHub к crates.io, нажмите кнопку « *Войти с GitHub* » в верхнюю панель меню и авторизуйте crates.io для доступа к вашей учетной записи. Затем вы запишете вас в crates.io, если все будет хорошо.

Затем вы должны найти свой **ключ API** , который можно найти, нажав на ваш аватар, выбрав « *Настройки учетной записи* » и скопировав строку, которая выглядит так:

```
cargo login abcdefghijklmnopqrstuvwxyz1234567890rust
```

Это должно быть вставлено в вашем терминале / командной строке, и должен аутентифицировать вас с вашей локальной `cargo` установкой.

Будьте осторожны с вашим ключом API - он **должен** храниться в секрете, как пароль, иначе ваши ящики могут быть угнаны!

Прочитайте грузовой онлайн: https://riptutorial.com/ru/rust/topic/1084/грузовой

глава 18: Дженерики

Examples

декларация

```
// Generic types are declared using the <T> annotation

struct GenericType<T> {
    pub item: T
}

enum QualityChecked<T> {
    Excellent(T),
    Good(T),
    // enum fields can be generics too
    Mediocre { product: T }
}
```

Конкретизация

```
// explicit type declaration
let some_value: Option<u32> = Some(13);

// implicit type declaration
let some_other_value = Some(66);
```

Множественные параметры типа

Типы обобщений могут иметь более одного типа параметров, например. `Result` определяется следующим образом:

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Ограниченные общие типы

```
// Only accept T and U generic types that also implement Debug
fn print_objects<T: Debug, U: Debug>(a: T, b: U) {
    println!("A: {:?} B: {:?}", a, b);
}

print_objects(13, 44);
// or annotated explicitly
print_objects::<usize, u16>(13, 44);
```

Оценки должны охватывать все виды использования этого типа. Добавление выполняется с помощью `std::ops::Add` trait, которая имеет входные и выходные параметры. `where T: std::ops::Add<u32,Output=U>` заявляет, что возможно `Add T` в `u32` , и это дополнение должно произвести тип `U`

```
fn try_add_one<T, U>(input_value: T) -> Result<U, String>
    where T: std::ops::Add<u32,Output=U>
{
    return Ok(input_value + 1);
}
```

По умолчанию подразумевается `Sized` граница. `?Sized` bound также позволяет использовать нестандартные типы.

Общие функции

Общие функции позволяют параметризировать некоторые или все их аргументы.

```
fn convert_values<T, U>(input_value: T) -> Result<U, String> {
  // Try and convert the value.
  // Actual code will require bounds on the types T, U to be able to do something with them.
}
```

Если компилятор не может вывести параметр типа, он может быть предоставлен вручную по вызову:

```
let result: Result<u32, String> = convert_value::<f64, u32>(13.5);
```

Прочитайте Дженерики онлайн: https://riptutorial.com/ru/rust/topic/1801/дженерики

глава 19: Документация

Вступление

Компилятор Rust имеет несколько удобных функций, позволяющих быстро и легко документировать ваш проект. Вы можете использовать компилятор lints для обеспечения документирования для каждой функции и иметь тесты, встроенные в ваши примеры.

Синтаксис

- /// Комментарий к внешней документации (относится к элементу ниже)
- //! Внутренний комментарий к документации (относится к прилагаемому элементу)
- cargo doc # Создает документацию для этого библиотечного ящика.
- load doc --open # Создает документацию для этого библиотечного ящика и открывает браузер.
- load doc -p CRATE # Создает документацию только для указанного ящика.
- load doc --no-deps # Создает документацию для этой библиотеки и никаких зависимостей.
- испытание груза # Выполняет модульные испытания и испытания документации.

замечания

Этот раздел «Книги ржавчины» может содержать полезную информацию о документах и документации.

Комментарии к документации могут быть применены для:

- Модули
- Структуры
- Перечисления
- методы
- функции
- Черты и методы

Examples

Документация Lints

Чтобы обеспечить документирование всех возможных элементов, вы можете использовать ссылку `missing_docs` для получения предупреждений / ошибок от компилятора. Чтобы получать предупреждения по всей библиотеке, поместите этот атрибут в свой файл `lib.rs` :

```
#![warn(missing_docs)]
```

Вы также можете получить ошибки для отсутствия документации с помощью этой строки:

```
#![deny(missing_docs)]
```

По умолчанию `missing_docs` разрешены, но вы можете явно разрешить им этот атрибут:

```
#![allow(missing_docs)]
```

Это может быть полезно разместить в одном модуле, чтобы разрешить отсутствующую документацию для одного модуля, но запретить его во всех других файлах.

Документация Комментарии

Rust предоставляет два типа комментариев документации: комментарии внутренней документации и внешние комментарии к документации. Примеры каждого из них приведены ниже.

Внутренняя документация Комментарии

```
mod foo {
    //! Inner documentation comments go *inside* an item (e.g. a module or a
    //! struct). They use the comment syntax //! and must go at the top of the
    //! enclosing item.
    struct Bar {
        pub baz: i64
        //! This is invalid. Inner comments must go at the top of the struct,
        //! and must not be placed after fields.
    }
}
```

Комментарии к внешней документации

```
/// Outer documentation comments go *outside* the item that they refer to.
/// They use the syntax /// to distinguish them from inner comments.
pub enum Test {
    Success,
    Fail(Error)
}
```

Условные обозначения

```
/// In documentation comments, you may use **Markdown**.
/// This includes `backticks` for code, *italics* and **bold**.
/// You can add headers in your documentation, like this:
/// # Notes
/// `Foo` is unsuitable for snafucating. Use `Bar` instead.
struct Foo {
    ...
}
```

```
/// It is considered good practice to have examples in your documentation
/// under an "Examples" header, like this:
/// # Examples
/// Code can be added in "fences" of 3 backticks.
///
/// ```
/// let bar = Bar::new();
/// ```
///
/// Examples also function as tests to ensure the examples actually compile.
/// The compiler will automatically generate a main() function and run the
/// example code as a test when cargo test is run.
struct Bar {
    ...
}
```

Тестирование документации

Код в комментариях к документации автоматически будет выполнен с помощью `cargo test` . Они известны как «тесты документации» и помогают обеспечить работу ваших примеров и не будут вводить пользователей в заблуждение.

Вы можете импортировать относительный из корня ящика (как если бы у `extern crate mycrate;` был скрытый `extern crate mycrate;` в верхней части примера)

```
/// ```
/// use mycrate::foo::Bar;
/// ```
```

Если ваш код может не выполняться должным образом в тесте документации, вы можете использовать атрибут `no_run` , например:

```
/// ```no_run
/// use mycrate::NetworkClient;
/// NetworkClient::login("foo", "bar");
/// ```
```

Вы также можете указать, что ваш код *должен* паниковать, например:

```
/// ```should_panic
/// unreachable!();
/// ```
```

Прочитайте Документация онлайн: https://riptutorial.com/ru/rust/topic/4865/документация

глава 20: Железная веб-платформа

Вступление

Iron - популярная веб-среда для Rust (основанная на библиотеке Hyper более низкого уровня), которая продвигает идею расширяемости посредством *промежуточного программного обеспечения* . Большая часть функциональности, необходимой для создания полезного веб-сайта, может быть найдена в промежуточном программном обеспечении Iron, а не в самой библиотеке.

Examples

Простой сервер «Hello»

Этот пример отправляет жестко запрограммированный ответ пользователю при отправке запроса сервера.

```
extern crate iron;

use iron::prelude::*;
use iron::status;

// You can pass the handler as a function or a closure. In this
// case, we've chosen a function for clarity.
// Since we don't care about the request, we bind it to _.
fn handler(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Hello, Stack Overflow")))
}

fn main() {
    Iron::new(handler).http("localhost:1337").expect("Server failed!")
}
```

При создании нового `Iron` Server в этом примере `expect` чтобы поймать любые ошибки с более описательным сообщением об ошибке. В производственных приложениях *обрабатывайте полученную ошибку* (см. Документацию по `http()`).

Установка железа

Добавьте эту зависимость в файл `Cargo.toml` :

```
[dependencies]
iron = "0.4.0"
```

Запустите `cargo build` и Cargo загрузит и установит указанную версию Iron.

Простая маршрутизация с использованием железа

Этот пример обеспечит базовую маршрутизацию через Iron.

Для начала вам нужно добавить зависимость Iron от вашего файла `Cargo.toml` .

```
[dependencies]
iron = "0.4.*"
```

Мы будем использовать собственную библиотеку Router от Iron. Для простоты проект Iron предоставляет эту библиотеку в составе библиотеки ядра Iron, что устраняет необходимость добавления ее в качестве отдельной зависимости. Затем мы ссылаемся как на библиотеку Iron, так и на библиотеку Router.

```
extern crate iron;
extern crate router;
```

Затем мы импортируем требуемые объекты, чтобы мы могли управлять маршрутизацией и возвращать ответ пользователю.

```
use iron::{Iron, Request, Response, IronResult};
use iron::status;
use router::{Router};
```

В этом примере мы сохраним это просто, написав логику маршрутизации в нашей функции `main()` . Конечно, по мере роста вашего приложения вы захотите разделить маршрутизацию, протоколирование, проблемы безопасности и другие области вашего веб-приложения. На данный момент это хорошая отправная точка.

```
fn main() {
    let mut router = Router::new();
    router.get("/", handler, "handler");
    router.get("/:query", query_handler, "query_handler");
```

Давайте перейдем к тому, чего мы достигли до сих пор. Наша программа в настоящее время создает новый объект Iron `Router` и прикрепляет два «обработчика» к двум типам URL-запроса: первый (`"/"`) является корнем нашего домена, а второй (`"/:query"`) - это любой путь под root.

Используя слово «запрос» с запятой перед словами «запрос», мы говорим Желу, чтобы взять эту часть пути URL как переменную и передать ее в наш обработчик.

Следующая строка кода - это то, как мы создаем экземпляр Iron, обозначая наш собственный объект `router` для управления нашими запросами URL. Домен и порт жестко закодированы в этом примере для простоты.

```
    Iron::new(router).http("localhost:3000").unwrap();
```

Затем мы объявляем две встроенные функции, которые являются нашими обработчиками, `handler` и `query_handler` . Они используются для демонстрации фиксированных URL-адресов и переменных URL-адресов.

Во второй функции мы берем переменную `"query"` из URL-адреса, хранящегося объектом запроса, и мы отправляем ее обратно пользователю в качестве ответа.

```
    fn handler(_: &mut Request) -> IronResult<Response> {
        Ok(Response::with((status::Ok, "OK")))
    }

    fn query_handler(req: &mut Request) -> IronResult<Response> {
        let ref query = req.extensions.get::<Router>()
            .unwrap().find("query").unwrap_or("/");
        Ok(Response::with((status::Ok, *query)))
    }
}
```

Если мы запустим этот пример, мы сможем просмотреть результат в веб-браузере на `localhost:3000` . Корень домена должен отвечать `"OK"` , и все, что находится под корнем, должно повторить путь назад.

Следующим шагом в этом примере может быть разделение маршрутизации и обслуживание статических страниц.

Прочитайте Железная веб-платформа онлайн: https://riptutorial.com/ru/rust/topic/8060/железная-веб-платформа

глава 21: Закрытие и лямбда-выражения

Examples

Простые лямбда-выражения

```
// A simple adder function defined as a lambda expression.
// Unlike with regular functions, parameter types often may be omitted because the
// compiler can infer their types
let adder = |a, b| a + b;
// Lambdas can span across multiple lines, like normal functions.
let multiplier = |a: i32, b: i32| {
    let c = b;
    let b = a;
    let a = c;
    a * b
};

// Since lambdas are anonymous functions, they can be called like other functions
println!("{}", adder(3, 5));
println!("{}", multiplier(3, 5));
```

Это отображает:

```
8
15
```

Простое закрытие

В отличие от обычных функций, лямбда-выражения могут захватывать их среды. Такие лямбды называются замыканиями.

```

// variable definition outside the lambda expression...
let lucky_number: usize = 663;

// but the our function can access it anyway, thanks to the closures
let print_lucky_number = || println!("{}", lucky_number);

// finally call the closure
print_lucky_number();
```

Это напечатает:

```
663
```

Lambdas с явными типами возврата

```
// lambda expressions can have explicitly annotated return types
let floor_func = |x: f64| -> i64 { x.floor() as i64 };
```

Прохождение лямбда

Поскольку лямбда-функции сами являются значениями, вы храните их в коллекциях, передаете их в функции и т. Д., Как и с другими значениями.

```
// This function takes two integers and a function that performs some operation on the two
arguments
fn apply_function<T>(a: i32, b: i32, func: T) -> i32 where T: Fn(i32, i32) -> i32 {
    // apply the passed function to arguments a and b
    func(a, b)
}

// let's define three lambdas, each operating on the same parameters
let sum = |a, b| a + b;
let product = |a, b| a * b;
let diff = |a, b| a - b;

// And now let's pass them to apply_function along with some arbitary values
println!("3 + 6 = {}", apply_function(3, 6, sum));
println!("-4 * 9 = {}", apply_function(-4, 9, product));
println!("7 - (-3) = {}", apply_function(7, -3, diff));
```

Это напечатает:

```
3 + 6 = 9
-4 * 9 = -36
7 - (-3) = 10
```

Возвращение лямбда от функций

Возвращение лямбда (или закрытий) из функций может быть сложным, потому что они реализуют черты, и поэтому их точный размер редко известен.

```
// Box in the return type moves the function from the stack to the heap
fn curried_adder(a: i32) -> Box<Fn(i32) -> i32> {
    // 'move' applies move semantics to a, so it can outlive this function call
    Box::new(move |b| a + b)
}

println!("3 + 4 = {}", curried_adder(3)(4));
```

Это отображает: `3 + 4 = 7`

Прочитайте Закрытие и лямбда-выражения онлайн: https://riptutorial.com/ru/rust/topic/1815/закрытие-и-лямбда-выражения

глава 22: Интерфейс внешних функций (FFI)

Синтаксис

- # [link (name = "snappy")] // внешнюю библиотеку, к которой нужно привязать (необязательно)

  extern {...} // список сигнатур функций в иностранной библиотеке

Examples

Вызов функции libc из ночной ржавчины

`libc` является « функцией gated » и может быть доступен только в ночных версиях Rust, пока он не станет стабильным.

```
#![feature(libc)]
extern crate libc;
use libc::pid_t;

#[link(name = "c")]
extern {
    fn getpid() -> pid_t;
}

fn main() {
    let x = unsafe { getpid() };
    println!("Process PID is {}", x);
}
```

Прочитайте Интерфейс внешних функций (FFI) онлайн:
https://riptutorial.com/ru/rust/topic/6140/интерфейс-внешних-функций--ffi-

глава 23: итераторы

Вступление

Итераторы - мощная языковая функция в Rust, описываемая признаком `Iterator` . Итераторы позволяют выполнять множество операций с типами, подобными коллекции, например `Vec<T>` , и они легко могут быть скомпонованы.

Examples

Адаптеры и потребители

Методы Итератора могут быть разбиты на две различные группы:

адаптеры

Адаптеры принимают итератор и возвращают другой итератор

```
//          Iterator  Adapter
//              |       |
let my_map = (1..6).map(|x| x * x);
println!("{:?}", my_map);
```

Выход

```
Map { iter: 1..6 }
```

Обратите внимание, что значения не были перечислены, что указывает на то, что итераторы не оцениваются с высокой вероятностью - итераторы «ленивы».

Потребители

Потребители берут итератор и возвращают нечто иное, чем итератор, потребляя итератор в процессе.

```
//                     Iterator  Adapter       Consumer
//                         |       |             |
let my_squares: Vec<_> = (1..6).map(|x| x * x).collect();
println!("{:?}", my_squares);
```

Выход

```
[1, 4, 9, 16, 25]
```

Другие примеры потребителей включают `find` , `fold` и `sum` .

```
let my_squared_sum: u32 = (1..6).map(|x| x * x).sum();
println!("{:?}", my_squared_sum);
```

Выход

```
55
```

Короткий тест на первичность

```
fn is_prime(n: u64) -> bool {
    (2..n).all(|divisor| n % divisor != 0)
}
```

Конечно, это не быстрый тест. Мы можем остановить тестирование с квадратным корнем из `n` :

```
(2..n)
    .take_while(|divisor| divisor * divisor <= n)
    .all(|divisor| n % divisor != 0)
```

Пользовательский итератор

```
struct Fibonacci(u64, u64);

impl Iterator for Fibonacci {
    type Item = u64;

    // The method that generates each item
    fn next(&mut self) -> Option<Self::Item> {
        let ret = self.0;
        self.0 = self.1;
        self.1 += ret;

        Some(ret) // since `None` is never returned, we have an infinite iterator
    }

    // Implementing the `next()` method suffices since every other iterator
    // method has a default implementation
}
```

Пример использования:

```
// the iterator method `take()` is an adapter which limits the number of items
// generated by the original iterator
for i in Fibonacci(0, 1).take(10) {
    println!("{}", i);
}
```

Прочитайте итераторы онлайн: https://riptutorial.com/ru/rust/topic/4657/итераторы

глава 24: Кортеж

Вступление

Самая тривиальная структура данных, после единственного значения, является кортежем.

Синтаксис

- (A, B, C) // трехстрочный (кортеж с тремя элементами), первый элемент которого имеет тип A, второй тип B и третий тип C
- (A, B) // двухстрочный, два элемента которого имеют тип A и B соответственно
- (A,) // один-кортеж (обратите внимание на завершающем `,`), который содержит только один элемент типа A
- () // пустой кортеж, который является как типом, так и единственным типом элемента

Examples

Типы кортежей и значения кортежа

Кортежи ржавчины, как и на большинстве других языков, представляют собой списки фиксированного размера, элементы которых могут быть разных типов.

```
// Tuples in Rust are comma-separated values or types enclosed in parentheses.
let _ = ("hello", 42, true);
// The type of a tuple value is a type tuple with the same number of elements.
// Each element in the type tuple is the type of the corresponding value element.
let _: (i32, bool) = (42, true);
// Tuples can have any number of elements, including one ..
let _: (bool,) = (true,);
// .. or even zero!
let _: () = ();
// this last type has only one possible value, the empty tuple ()
// this is also the type (and value) of the return value of functions
// that do not return a value ..
let _: () = println!("hello");
// .. or of expressions with no value.
let mut a = 0;
let _: () = if true { a += 1; };
```

Соответствие значений кортежа

Программы ржавчины широко используют шаблоны для деконструирования значений, независимо от того, используют ли они `match` , `if let` или деконструируют шаблоны `let` . Кортежи могут быть деконструированы, как вы могли бы ожидать, используя `match`

```
fn foo(x: (&str, isize, bool)) {
```

```
    match x {
        (_, 42, _) => println!("it's 42"),
        (_, _, false) => println!("it's not true"),
        _ => println!("it's something else"),
    }
}
```

или `if let`

```
fn foo(x: (&str, isize, bool)) {
    if let (_, 42, _) = x {
        println!("it's 42");
    } else {
        println!("it's something else");
    }
}
```

вы также можете связать внутри кортежа, используя `let` -deconstruction

```
fn foo(x: (&str, isize, bool)) {
    let (_, n, _) = x;
    println!("the number is {}", n);
}
```

Внутри кортежей

Чтобы напрямую обращаться к элементам кортежа, вы можете использовать формат `.n` для доступа к `n` му элементу

```
let x = ("hello", 42, true);
assert_eq!(x.0, "hello");
assert_eq!(x.1, 42);
assert_eq!(x.2, true);
```

Вы также можете частично выйти из кортежа

```
let x = (String::from("hello"), 42);
let (s, _) = x;
let (_, n) = x;
println!("{}, {}", s, n);
// the following would fail however, since x.0 has already been moved
// let foo = x.0;
```

основы

Кортеж - это просто конкатенация нескольких значений:

- возможных типов
- число и типы которых известны статически

Например, `(1, "Hello")` является 2-мя элементами, состоящими из `i32` и а `&str` , и его тип

обозначается как `(i32, &'static str)` аналогично его значению.

Чтобы получить доступ к элементу кортежа, просто просто используется его индекс:

```
let tuple =  (1, "Hello");
println!("First element: {}, second element: {}", tuple.0, tuple.1);
```

Поскольку кортеж встроен, также возможно использовать сопоставление образцов по кортежам:

```
match (1, "Hello") {
    (i, _) if i < 0 => println!("Negative integer: {}", i),
    (_, s) => println!("{} World", s),
}
```

Особые случаи

Корневой элемент 0: `()` также называется *единицей* , *типом единицы* или *одноточечным типом* и используется для обозначения отсутствия значимых значений. Это возвращаемый тип функции по умолчанию (когда `->` не указано). *См. Также: Какой тип «type ()» в Rust?* ,

1-элементный кортеж: `(a,)` с конечной запятой обозначает 1 элементный кортеж. Форма без запятой `(a)` интерпретируется как выражение, заключенное в круглые скобки, и оценивается как `a` .

И хотя мы и находимся в этом, конечные запятые всегда принимаются: `(1, "Hello",)` .

Ограничения

Сегодня язык ржавчины не поддерживает *вариации* , кроме кортежей. Поэтому невозможно просто реализовать признак для всех кортежей, и в результате стандартные черты реализуются только для кортежей до ограниченного числа элементов (сегодня до 12 включительно). Поддерживаются кортежи с большим количеством элементов, но не реализуют стандартные черты (хотя вы можете реализовать свои собственные черты).

Это ограничение, мы надеемся, будет снято в будущем.

Распаковка кортежей

```
// It's possible to unpack tuples to assign their inner values to variables
let tup = (0, 1, 2);
// Unpack the tuple into variables a, b, and c
let (a, b, c) = tup;

assert_eq!(a, 0);
assert_eq!(b, 1);

// This works for nested data structures and other complex data types
```

```
let complex = ((1, 2), 3, Some(0));

let (a, b, c) = complex;
let (aa, ab) = a;

assert_eq!(aa, 1);
assert_eq!(ab, 2);
```

Прочитайте Кортеж онлайн: https://riptutorial.com/ru/rust/topic/3941/кортеж

глава 25: макрос

замечания

Прохождение макросов можно найти на языке *программирования ржавчины* (aka the Book)
.

Examples

Руководство

Макросы позволяют нам абстрагировать синтаксические шаблоны, которые повторяются много раз. Например:

```
/// Computes `a + b * c`. If any of the operation overflows, returns `None`.
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = match b.checked_mul(c) {
        Some(p) => p,
        None => return None,
    };
    let sum = match a.checked_add(product) {
        Some(s) => s,
        None => return None,
    };
    Some(sum)
}
```

Заметим , что два `match` заявления очень похожи: оба они имеют один и тот же шаблон

```
match expression {
    Some(x) => x,
    None => return None,
}
```

Представьте, что мы представляем вышеприведенный шаблон как `try_opt!(expression)` , тогда мы могли бы переписать функцию только в 3 строки:

```
fn checked_fma(a: u64, b: u64, c: u64) -> Option<u64> {
    let product = try_opt!(b.checked_mul(c));
    let sum = try_opt!(a.checked_add(product));
    Some(sum)
}
```

`try_opt!` не может написать функцию, потому что функция не поддерживает раннее возвращение. Но мы могли бы сделать это с помощью макроса - всякий раз, когда у нас есть синтаксические шаблоны, которые невозможно представить с помощью функции, мы можем попытаться использовать макрос.

Мы определяем макрос с помощью `macro_rules!` синтаксис:

```
macro_rules! try_opt {
//                  ^ note: no `!` after the macro name
    ($e:expr) => {
//   ^~~~~~~ The macro accepts an "expression" argument, which we call `$e`.
//           All macro parameters must be named like `$xxxxx`, to distinguish from
//           normal tokens.
        match $e {
//            ^~ The input is used here.
            Some(x) => x,
            None => return None,
        }
    }
}
```

Это оно! Мы создали наш первый макрос.

(Попробуйте в Rust Playground)

Создание макроса HashSet

```
// This example creates a macro `set!` that functions similarly to the built-in
// macro vec!

use std::collections::HashSet;

macro_rules! set {
    ( $( $x:expr ),* ) => {  // Match zero or more comma delimited items
        {
            let mut temp_set = HashSet::new();  // Create a mutable HashSet
            $(
                temp_set.insert($x); // Insert each item matched into the HashSet
            )*
            temp_set // Return the populated HashSet
        }
    };
}

// Usage
let my_set = set![1, 2, 3, 4];
```

Рекурсия

Макрос может вызывать себя, как функция рекурсии:

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

Пойдем, хотя расширение `sum!(1, 2, 3)` :

```
   sum!(1, 2, 3)
```

```
//      ^  ^~~~
//     $a  $rest
=> 1 + sum!(2, 3)
//           ^  ^
//          $a  $rest
=> 1 + (2 + sum!(3))
//                ^
//                $base
=> 1 + (2 + (3))
```

Предел рекурсии

Когда компилятор слишком сильно расширяет макросы, он откажется. По умолчанию компилятор завершится с ошибкой после расширения макросов до 64 уровней, поэтому, например, следующее расширение приведет к сбою:

```
sum!(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
     21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,
     41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62)

// error: recursion limit reached while expanding the macro `sum`
//  --> <anon>:3:46
// 3 |>          ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
//   |>                                              ^^^^^^^^^^^^^^^^
```

Когда достигнут предел рекурсии, вы должны рассмотреть возможность реорганизации своего макроса, например

- Может быть, рекурсия может быть заменена повторением?
- Может быть, формат ввода может быть изменен на что-то менее фантастическое, поэтому нам не нужна рекурсия, чтобы соответствовать ему?

Если есть какая-то законная причина, недостаточно 64 уровней, вы всегда можете увеличить лимит ящика, ссылаясь на макрос с атрибутом:

```
#![recursion_limit="128"]
//                  ^~~ set the recursion limit to 128 levels deep.
```

Несколько шаблонов

Макрос может создавать различные выходные данные для разных шаблонов ввода:

```
/// The `sum` macro may be invoked in two ways:
///
///     sum!(iterator)
///     sum!(1234, iterator)
///
macro_rules! sum {
    ($iter:expr) => {    // This branch handles the `sum!(iterator)` case
        $iter.fold(0, |a, b| a + *b)
```

```
    };
//   ^ use `;` to separate each branch
    ($start:expr, $iter:expr) => {  // This branch handles the `sum!(1234, iter)` case
        $iter.fold($start, |a, b| a + *b)
    };
}

fn main() {
    assert_eq!(10, sum!([1, 2, 3, 4].iter()));
    assert_eq!(23, sum!(6, [2, 5, 9, 1].iter()));
}
```

Спецификаторы фрагментов - вид шаблонов

В `$e:expr` `expr` называется *спецификатором фрагмента* . Он сообщает парсеру, какие маркеры ожидает параметр `$e` . Rust предоставляет множество спецификаторов фрагментов, что позволяет вводить данные очень гибкими.

Тендерный	Описание	Примеры
`ident`	Идентификатор	`x` , `foo`
`path`	Квалифицированное имя	`std::collection::HashSet` , `Vec::new`
`ty`	Тип	`i32` , `&T` , `Vec<(char, String)>`
`expr`	выражение	`2+2` , `f(42)` , `if true { 1 } else { 2 }`
`pat`	Шаблон	`_` , `c @ 'a' ... 'z'` , `(true, &x)` , `Badger { age, .. }`
`stmt`	утверждение	`let x = 3` , `return 42`
`block`	Блок с разделителями	`{ foo(); bar(); }` , `{ x(); y(); z() }`
`item`	Вещь	`fn foo() {}` , `struct Bar;` , `use std::io;`
`meta`	Внутри атрибута	`cfg!(windows)` , `doc="comment"`
`tt`	Токен	`+` , `foo` , `5` , `[?!(???)]`

Обратите внимание, что комментарий doc comment `/// comment` рассматривается так же, как `#[doc="comment"]` для макроса.

```
macro_rules! declare_const_option_type {
    (
        $(#[$attr:meta])*
        const $name:ident: $ty:ty as optional;
    ) => {
```

```
        $(#[$attr])*
        const $name: Option<$ty> = None;
    }
}

declare_const_option_type! {
    /// some doc comment
    const OPT_INT: i32 as optional;
}

// The above will be expanded to:
#[doc="some doc comment"]
const OPT_INT: Option<i32> = None;
```

Следуйте установленному

Некоторым спецификаторам фрагментов требуется токен, следующий за ним, который должен быть одним из ограниченного набора, называемым «follow set». Это позволяет немного гибкости для синтаксиса Rust, чтобы развиваться без нарушения существующих макросов.

Тендерный	Следуйте установленному
`expr` , `stmt`	`=>` , `;`
`ty` , `path`	`=>` , `=` | `;` `:` `>` `[` `{` `as` , `where`
`pat`	`=>` , `=` | `if` `in`
`ident` , `block` , `item` , `meta` , `tt`	*любой токен*

```
macro_rules! invalid_macro {
    ($e:expr + $f:expr) => { $e + $f };
//           ^
//           `+` is not in the follow set of `expr`,
//           and thus the compiler will not accept this macro definition.
    ($($e:expr)/+) => { $($e)/+ };
//              ^
//              The separator `/` is not in the follow set of `expr`
//              and thus the compiler will not accept this macro definition.
}
```

Экспорт и импорт макросов

Экспорт макроса, чтобы другие модули могли его использовать:

```
    #[macro_export]
//  ^~~~~~~~~~~~~~~ Think of it as `pub` for macros.
    macro_rules! my_macro { (..) => {..} }
```

Использование макросов из других ящиков или модулей:

```
   #[macro_use] extern crate lazy_static;
// ^~~~~~~~~~~~ Must add this in order to use macros from other crates

   #[macro_use] mod macros;
// ^~~~~~~~~~~~ The same for child modules.
```

Отладка макросов

(Все они нестабильны и поэтому могут использоваться только из ночного компилятора).

log_syntax! ()

```
#![feature(log_syntax)]

macro_rules! logged_sum {
    ($base:expr) => {
        { log_syntax!(base = $base); $base }
    };
    ($a:expr, $($rest:expr),+) => {
        { log_syntax!(a = $a, rest = $($rest),+); $a + logged_sum!($($rest),+) }
    };
}

const V: u32 = logged_sum!(1, 2, 3);
```

Во время компиляции он выведет на stdout следующее:

a = 1, остаток = 2, 3
a = 2, остаток = 3
base = 3

- Довольно расширенный

Запустите компилятор с помощью:

```
rustc -Z unstable-options --pretty expanded filename.rs
```

Это расширит все макросы, а затем распечатает расширенный результат в stdout, например, выше, вероятно, будет выводиться:

```
#![feature(prelude_import)]
#![no_std]
#![feature(log_syntax)]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
```

```
extern crate std as std;


const V: u32 = { false; 1 + { false; 2 + { false; 3 } } };
```

(Это похоже на флаг `-E` в компиляторах `gcc` и `clang` .)

Прочитайте макрос онлайн: https://riptutorial.com/ru/rust/topic/1031/макрос

глава 26: Массивы, векторы и срезы

Examples

Массивы

Массив - это список объектов с одним стеком, заданный стеками.

Массивы обычно создаются путем включения списка элементов определенного типа между квадратными скобками. Тип массива обозначается специальным синтаксисом: `[T; N]` где `T` - тип его элементов, а `N` их счет, оба из которых должны быть известны во время компиляции.

Например, `[4u64, 5, 6]` представляет собой 3-элементный массив типа `[u64; 3]` .
Примечание: `5` и `6` как предполагается, имеют тип `u64` .

пример

```
fn main() {
    // Arrays have a fixed size.
    // All elements are of the same type.
    let array = [1, 2, 3, 4, 5];

    // Create an array of 20 elements where all elements are the same.
    // The size should be a compile-time constant.
    let ones = [1; 20];

    // Get the length of an array.
    println!("Length of ones: {}", ones.len());

    // Access an element of an array.
    // Indexing starts at 0.
    println!("Second element of array: {}", array[1]);

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", array[5]);
}
```

Ограничения

Совпадение шаблонов на массивах (или срезах) не поддерживается в стабильном Rust (см. # 23121 и шаблоны срезов).

Ржавчина не поддерживает типичность цифр типа (см. RFC # 1657). Поэтому невозможно просто реализовать признак для всех массивов (всех размеров). В результате стандартные черты реализуются только для массивов с ограниченным количеством элементов (последний проверен, до 32 включительно). Массивы с большим количеством элементов поддерживаются, но не реализуют стандартные черты (см. Документы).

Эти ограничения, мы надеемся, будут сняты в будущем.

векторы

Вектор - это, по существу, указатель на список, выделенный динамическим размером объектов одного типа.

пример

```
fn main() {
    // Create a mutable empty vector
    let mut vector = Vec::new();

    vector.push(20);
    vector.insert(0, 10); // insert at the beginning

    println!("Second element of vector: {}", vector[1]); // 20

    // Create a vector using the `vec!` macro
    let till_five = vec![1, 2, 3, 4, 5];

    // Create a vector of 20 elements where all elements are the same.
    let ones = vec![1; 20];

    // Get the length of a vector.
    println!("Length of ones: {}", ones.len());

    // Run-time bounds-check.
    // This panics with 'index out of bounds: the len is 5 but the index is 5'.
    println!("Non existant element of array: {}", till_five[5]);
}
```

Ломтики

Срезы - это виды в список объектов и имеют тип `[T]` , указывающий кусочек объектов с типом `T`

Срез - это нестандартный тип , поэтому его можно использовать только по указателю. *(Строковая мировая аналогия: `str` , называемый строковым срезом, также не поддерживается.)*

Массивы набираются на срезы, и векторы могут быть разыменованы на срезы. Поэтому

методы среза могут применяться к обоим из них. *(Строковая мировая аналогия: `str` - `String`, что `[T]` - `Vec<T>`.)*

```
fn main() {
    let vector = vec![1, 2, 3, 4, 5, 6, 7, 8];
    let slice = &vector[3..6];
    println!("length of slice: {}", slice.len()); // 3
    println!("slice: {:?}", slice); // [4, 5, 6]
}
```

Прочитайте Массивы, векторы и срезы онлайн: https://riptutorial.com/ru/rust/topic/5004/массивы--векторы-и-срезы

глава 27: Модули

Синтаксис

- mod *modname* ; // Поиск модуля в *modname* .rs или *modname* /mod.rs в том же каталоге
- mod *modname* { *block* }

Examples

Дерево модулей

файлы:

```
- example.rs (root of our modules tree, generally named lib.rs or main.rs when using Cargo)
- first.rs
- second/
  - mod.rs
  - sub.rs
```

Модули:

```
- example        -> example
  - first        -> example::first
  - second       -> example::second
    - sub        -> example::second::sub
  - third        -> example::third
```

`example.rs`

```
pub mod first;
pub mod second;
pub mod third {
    ...
}
```

Модуль `second` должен быть объявлен в `example.rs` файл , как его родитель `example` , а не, например, `first` и , таким образом , не может быть объявлен в `first.rs` или другой файл в каталоге того же уровень

`second/mod.rs`

```
pub mod sub;
```

Атрибут # [путь]

Атрибут Rust `#[path]` может использоваться для указания пути к поиску определенного модуля, если он не находится в стандартном расположении. Это, как правило,

обескураживает , потому что это делает иерархию модулей хрупкой и позволяет легко разбивать сборку, перемещая файл в совершенно другой каталог.

```
#[path="../path/to/module.rs"]
mod module;
```

Имена в коде с именами в `use`

Синтаксис с двойной колонкой имен в операторе `use` похож на имена, используемые в другом месте кода, но значение этих путей различно.

Имена в инструкции `use` по умолчанию интерпретируются как абсолютные, начиная с корня ящика. Имена в другом месте кода относятся к текущему модулю.

Заявление:

```
use std::fs::File;
```

имеет то же значение в основном файле ящика, а также в модулях. С другой стороны, имя функции, такое как `std::fs::File::open()` будет относиться к стандартной библиотеке Rust только в основном файле ящика, потому что имена в коде интерпретируются относительно текущего модуля.

```
fn main() {
    std::fs::File::open("example"); // OK
}

mod my_module {
   fn my_fn() {
       // Error! It means my_module::std::fs::File::open()
       std::fs::File::open("example");

       // OK. `::` prefix makes it absolute
       ::std::fs::File::open("example");

       // OK. `super::` reaches out to the parent module, where `std` is present
       super::std::fs::File::open("example");
   }
}
```

Чтобы заставить `std::…` имена вести себя везде так же, как в корневом ящике, вы можете добавить:

```
use std;
```

И наоборот, вы можете `use` пути `use` относительно префикса их с помощью `self` или `super` keywords:

```
 use self::my_module::my_fn;
```

Доступ к родительскому модулю

Иногда бывает полезно импортировать функции и структуры относительно без необходимости `use` что-то с его абсолютным путем в вашем проекте. Для этого вы можете использовать модуль `super` , например:

```
fn x() -> u8 {
    5
}

mod example {
    use super::x;

    fn foo() {
        println!("{}", x());
    }
}
```

Вы можете использовать `super` несколько раз , чтобы достичь «прародителя» вашего текущего модуля, но вы должны быть осторожны с введением проблем читаемости , если вы используете `super` слишком много раз в одном импорта.

Экспорт и видимость

Структура каталога:

```
yourproject/
    Cargo.lock
    Cargo.toml
    src/
        main.rs
        writer.rs
```

main.rs

```
// This is import from writer.rs
mod writer;

fn main() {
    // Call of imported write() function.
    writer::write()

    // BAD
    writer::open_file()
}
```

writer.rs

```
// This function WILL be exported.
pub fn write() {}

// This will NOT be exported.
```

```
fn open_file() {}
```

Организация базового кода

Давайте посмотрим, как мы можем организовать код, когда кодовая база становится больше.

01. Функции

```
fn main() {
  greet();
}

fn greet() {
  println!("Hello, world!");
}
```

02. Модули - в том же файле

```
fn main() {
  greet::hello();
}

mod greet {
  // By default, everything inside a module is private
  pub fn hello() { // So function has to be public to access from outside
    println!("Hello, world!");
  }
}
```

03. Модули - в другом файле в том же каталоге

При перемещении некоторого кода в новый файл нет необходимости обертывать код в объявлении `mod` . Сам файл действует как модуль.

```
// ↳ main.rs
mod greet; // import greet module

fn main() {
  greet::hello();
}
```

```
// ↳ greet.rs
pub fn hello() { // function has to be public to access from outside
  println!("Hello, world!");
}
```

> При перемещении какого-либо кода в новый файл, если этот код был завернут из объявления `mod` , это будет дополнительный модуль файла.

```
// ↳ main.rs
```

```
mod greet;

fn main() {
  greet::hello::greet();
}
```

```
// ↳ greet.rs
pub mod hello { // module has to be public to access from outside
  pub fn greet() { // function has to be public to access from outside
    println!("Hello, world!");
  }
}
```

04. Модули - в другом файле в другом каталоге

При перемещении некоторого кода в новый файл в другой каталог, сам каталог действует как модуль. И `mod.rs` в корне модуля является точкой входа в модуль каталога. Все остальные файлы в этом каталоге действуют как вспомогательный модуль этого каталога.

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello();
}
```

```
// ↳ greet/mod.rs
pub fn hello() {
  println!("Hello, world!");
}
```

Когда в корне модуля есть несколько файлов,

```
// ↳ main.rs
mod greet;

fn main() {
  greet::hello_greet()
}
```

```
// ↳ greet/mod.rs
mod hello;

pub fn hello_greet() {
  hello::greet()
}
```

```
// ↳ greet/hello.rs
pub fn greet() {
  println!("Hello, world!");
}
```

05. Модули - с `self`

```
fn main() {
  greet::call_hello();
}

mod greet {
  pub fn call_hello() {
    self::hello();
  }

  fn hello() {
    println!("Hello, world!");
  }
}
```

06. Модули - С `super`

1. Когда вы хотите получить доступ к корневой функции изнутри модуля,

```
fn main() {
  dash::call_hello();
}

fn hello() {
  println!("Hello, world!");
}

mod dash {
  pub fn call_hello() {
    super::hello();
  }
}
```

2. Когда вы хотите получить доступ к функции во внешнем / родительском модуле изнутри вложенного модуля,

```
fn main() {
  outer::inner::call_hello();
}

mod outer {

  pub fn hello() {
    println!("Hello, world!");
  }

  mod inner {
    pub fn call_hello() {
      super::hello();
    }
  }

}
```

07. Модули - с `use`

1. Если вы хотите связать полный путь с новым именем,

```
use greet::hello::greet as greet_hello;

fn main() {
  greet_hello();
}

mod greet {
  pub mod hello {
    pub fn greet() {
      println!("Hello, world!");
    }
  }
}
```

2. Когда вы хотите использовать содержимое уровня области ящика

```
fn main() {
  user::hello();
}

mod greet {
  pub mod hello {
    pub fn greet() {
      println!("Hello, world!");
    }
  }
}

mod user {
  use greet::hello::greet as call_hello;

  pub fn hello() {
    call_hello();
  }
}
```

Прочитайте Модули онлайн: https://riptutorial.com/ru/rust/topic/2528/модули

глава 28: Небезопасные рекомендации

Вступление

Объясните, почему некоторые вещи отмечены `unsafe` в Rust, и почему нам может понадобиться использовать этот люк в определенных (редких) ситуациях.

Examples

Гонки данных

Гонки данных происходят, когда часть памяти обновляется одной стороной, а другая пытается ее прочитать или обновить одновременно (без синхронизации между ними). Давайте рассмотрим классический пример гонки данных, используя общий счетчик.

```
use std::cell::UnsafeCell;
use std::sync::Arc;
use std::thread;

// `UnsafeCell` is a zero-cost wrapper which informs the compiler that "what it
// contains might be shared mutably." This is used only for static analysis, and
// gets optimized away in release builds.
struct RacyUsize(UnsafeCell<usize>);

// Since UnsafeCell is not thread-safe, the compiler will not auto-impl Sync for
// any type containig it. And manually impl-ing Sync is "unsafe".
unsafe impl Sync for RacyUsize {}

impl RacyUsize {
    fn new(v: usize) -> RacyUsize {
        RacyUsize(UnsafeCell::new(v))
    }

    fn get(&self) -> usize {
        // UnsafeCell::get() returns a raw pointer to the value it contains
        // Dereferencing a raw pointer is also "unsafe"
        unsafe { *self.0.get() }
    }

    fn set(&self, v: usize) { // note: `&self` and not `&mut self`
        unsafe { *self.0.get() = v }
    }
}

fn main() {
    let racy_num = Arc::new(RacyUsize::new(0));

    let mut handlers = vec![];
    for _ in 0..10 {
        let racy_num = racy_num.clone();
        handlers.push(thread::spawn(move || {
            for i in 0..1000 {
```

```
                if i % 200 == 0 {
                    // give up the time slice to scheduler
                    thread::yield_now();
                    // this is needed to interleave the threads so as to observe
                    // data race, otherwise the threads will most likely be
                    // scheduled one after another.
                }

                // increment by one
                racy_num.set(racy_num.get() + 1);
            }
        }));
    }

    for th in handlers {
        th.join().unwrap();
    }

    println!("{}", racy_num.get());
}
```

При работе на многоядерном процессоре выход почти всегда будет меньше `10000` (10 потоков × 1000).

В этом примере гонка данных породила логически неправильную, но все еще значимую ценность. Это связано с тем, что в гонке участвовало только одно слово, и, таким образом, обновление не могло частично его изменить. Но расы данных в целом могут приводить к поврежденным значениям, которые являются недопустимыми для типа (тип небезопасным), когда объект, который ведется, охватывает несколько слов и / или создает значения, указывающие на недопустимые ячейки памяти (небезопасные ячейки памяти), когда указатели задействованы.

Тем не менее, тщательное использование атомных примитивов может позволить построить очень эффективные структуры данных, которые могут внутренне нуждаться в выполнении некоторых из этих «опасных» операций для выполнения действий, которые не статически проверяются системой типа Rust, но правильны в целом (т.е. для создания безопасного абстракция).

Прочитайте Небезопасные рекомендации онлайн: https://riptutorial.com/ru/rust/topic/6018/небезопасные-рекомендации

глава 29: Обработка ошибок

Вступление

Rust использует значения `Result<T, E>` чтобы указать восстановимые ошибки во время выполнения. Неустранимые ошибки вызывают панику, которая является отдельной темой.

замечания

Подробности обработки ошибок описаны на языке *программирования ржавчины* (aka the Book)

Examples

Общие методы результата

```
use std::io::{Read, Result as IoResult};
use std::fs::File;

struct Config(u8);

fn read_config() -> IoResult<String> {
    let mut s = String::new();
    let mut file = File::open(&get_local_config_path())
        // or_else closure is invoked if Result is Err.
        .or_else(|_| File::open(&get_global_config_path()))?;
    // Note: In `or_else`, the closure should return a Result with a matching
    //       Ok type, whereas in `and_then`, the returned Result should have a
    //       matching Err type.
    let _ = file.read_to_string(&mut s)?;
    Ok(s)
}

struct ParseError;

fn parse_config(conf_str: String) -> Result<Config, ParseError> {
    // Parse the config string...
    if conf_str.starts_with("bananas") {
        Err(ParseError)
    } else {
        Ok(Config(42))
    }
}

fn run() -> Result<(), String> {
    // Note: The error type of this function is String. We use map_err below to
    //       make the error values into String type
    let conf_str = read_config()
        .map_err(|e| format!("Failed to read config file: {}", e))?;
    // Note: Instead of using `?` above, we can use `and_then` to wrap the let
    //       expression below.
```

```
    let conf_val = parse_config(conf_str)
        .map(|Config(v)| v / 2) // map can be used to map just the Ok value
        .map_err(|_| "Failed to parse the config string!".to_string())?;

    // Run...

    Ok(())
}

fn main() {
    match run() {
        Ok(_) => println!("Bye!"),
        Err(e) => println!("Error: {}", e),
    }
}

fn get_local_config_path() -> String {
    let user_config_prefix = "/home/user/.config";
    // code to get the user config directory
    format!("{}/my_app.rc", user_config_prefix)
}

fn get_global_config_path() -> String {
    let global_config_prefix = "/etc";
    // code to get the global config directory
    format!("{}/my_app.rc", global_config_prefix)
}
```

Если конфигурационные файлы не существуют, это выводит:

```
Error: Failed to read config file: No such file or directory (os error 2)
```

Если синтаксический анализ не завершился, это приведет к:

```
Error: Failed to parse the config string!
```

Примечание. По мере роста проекта будет громоздок обрабатывать ошибки с помощью этих основных методов (docs), не теряя информацию о происхождении и пути распространения ошибок. Кроме того, некорректная практика заключается в том, чтобы преждевременно преобразовывать ошибки в строки, чтобы обрабатывать несколько типов ошибок, как показано выше. Гораздо лучший способ - использовать `error-chain` в ящике.

Пользовательские типы ошибок

```
use std::error::Error;
use std::fmt;
use std::convert::From;
use std::io::Error as IoError;
use std::str::Utf8Error;

#[derive(Debug)] // Allow the use of "{:?}" format specifier
enum CustomError {
    Io(IoError),
    Utf8(Utf8Error),
```

```
    Other,
}

// Allow the use of "{}" format specifier
impl fmt::Display for CustomError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            CustomError::Io(ref cause) => write!(f, "I/O Error: {}", cause),
            CustomError::Utf8(ref cause) => write!(f, "UTF-8 Error: {}", cause),
            CustomError::Other => write!(f, "Unknown error!"),
        }
    }
}

// Allow this type to be treated like an error
impl Error for CustomError {
    fn description(&self) -> &str {
        match *self {
            CustomError::Io(ref cause) => cause.description(),
            CustomError::Utf8(ref cause) => cause.description(),
            CustomError::Other => "Unknown error!",
        }
    }

    fn cause(&self) -> Option<&Error> {
        match *self {
            CustomError::Io(ref cause) => Some(cause),
            CustomError::Utf8(ref cause) => Some(cause),
            CustomError::Other => None,
        }
    }
}

// Support converting system errors into our custom error.
// This trait is used in `try!`.
impl From<IoError> for CustomError {
    fn from(cause: IoError) -> CustomError {
        CustomError::Io(cause)
    }
}
impl From<Utf8Error> for CustomError {
    fn from(cause: Utf8Error) -> CustomError {
        CustomError::Utf8(cause)
    }
}
```

Итерация по причинам

Для целей отладки часто бывает полезно найти основную причину ошибки. Чтобы проверить значение ошибки, реализующее `std::error::Error` :

```
use std::error::Error;

let orig_error = call_returning_error();

// Use an Option<&Error>. This is the return type of Error.cause().
let mut err = Some(&orig_error as &Error);
```

```
// Print each error's cause until the cause is None.
while let Some(e) = err {
    println!("{}", e);
    err = e.cause();
}
```

Основная отчетность об ошибках и обработка

`Result<T, E>` - это тип `enum` который имеет два варианта: `Ok(T)` указывающий успешное выполнение со значимым результатом типа `T` и `Err(E)` указывающее на появление непредвиденной ошибки во время выполнения, описываемое значением типа `E` ,

```
enum DateError {
    InvalidDay,
    InvalidMonth,
}

struct Date {
    day: u8,
    month: u8,
    year: i16,
}

fn validate(date: &Date) -> Result<(), DateError> {
    if date.month < 1 || date.month > 12 {
        Err(DateError::InvalidMonth)
    } else if date.day < 1 || date.day > 31 {
        Err(DateError::InvalidDay)
    } else {
        Ok(())
    }
}

fn add_days(date: Date, days: i32) -> Result<Date, DateError> {
    validate(&date)?; // notice `?` -- returns early on error
    // the date logic ...
    Ok(date)
}
```

Также см. Документы для более подробной информации `?` оператор.

Стандартная библиотека содержит `Error` признак , который все типы ошибок рекомендуется реализовать. Ниже приведен пример реализации.

```
use std::error::Error;
use std::fmt;

#[derive(Debug)]
enum DateError {
    InvalidDay(u8),
    InvalidMonth(u8),
}

impl fmt::Display for DateError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
```

```
            &DateError::InvalidDay(day) => write!(f, "Day {} is outside range!", day),
            &DateError::InvalidMonth(month) => write!(f, "Month {} is outside range!", month),
        }
    }
}

impl Error for DateError {
    fn description(&self) -> &str {
        match self {
            &DateError::InvalidDay(_) => "Day is outside range!",
            &DateError::InvalidMonth(_) => "Month is outside range!",
        }
    }

    // cause method returns None by default
}
```

Примечание. Как правило, параметр `Option<T>` не должен использоваться для сообщения об ошибках. `Option<T>` указывает на ожидаемую возможность отсутствия значения и единственную прямую причину этого. Напротив, `Result<T, E>` используется для сообщения о неожиданных ошибках во время выполнения, и особенно когда существует несколько способов сбоев различения между ними. Кроме того, `Result<T, E>` используется только как возвращаемые значения. (Старая дискуссия.)

Прочитайте Обработка ошибок онлайн: https://riptutorial.com/ru/rust/topic/1762/обработка-ошибок

глава 30: Обработка сигналов

замечания

У Rust нет надлежащего идиоматического и безопасного способа лидировать с сигналами ОС, но есть некоторые ящики, которые обеспечивают обработку сигналов, но они очень *экспериментальные* и *небезопасные,* поэтому будьте осторожны при их использовании.

Однако в репозитории rust-lang / rfcs обсуждается вопрос о внедрении встроенной обработки сигналов для ржавчины.

Обсуждение RFC: https://github.com/rust-lang/rfcs/issues/1368

Examples

Обработка сигналов с чан-сигнальным ящиком

Чан-сигнальный ящик обеспечивает решение для обработки сигнала ОС с использованием каналов, хотя этот ящик является **экспериментальным** и его следует использовать **осторожно** .

Пример, взятый из BurntSushi / chan-signal .

```
#[macro_use]
extern crate chan;
extern crate chan_signal;

use chan_signal::Signal;

fn main() {
    // Signal gets a value when the OS sent a INT or TERM signal.
    let signal = chan_signal::notify(&[Signal::INT, Signal::TERM]);
    // When our work is complete, send a sentinel value on `sdone`.
    let (sdone, rdone) = chan::sync(0);
    // Run work.
    ::std::thread::spawn(move || run(sdone));

    // Wait for a signal or for work to be done.
    chan_select! {
        signal.recv() -> signal => {
            println!("received signal: {:?}", signal)
        },
        rdone.recv() => {
            println!("Program completed normally.");
        }
    }
}

fn run(_sdone: chan::Sender<()>) {
    println!("Running work for 5 seconds.");
```

```
    println!("Can you send a signal quickly enough?");
    // Do some work.
    ::std::thread::sleep_ms(5000);

    // _sdone gets dropped which closes the channel and causes `rdone`
    // to unblock.
}
```

Обработка сигналов с помощью nix-ящика.

Ячейка nix предоставляет UNIX Rust API для обработки сигналов, однако для этого требуется использование **небезопасной** ржавчины, поэтому вы должны быть **осторожны** .

```
use nix::sys::signal;

extern fn handle_sigint(_:i32) {
    // Be careful here...
}

fn main() {
    let sig_action = signal::SigAction::new(handle_sigint,
                                            signal::SockFlag::empty(),
                                            signal::SigSet::empty());
    signal::sigaction(signal::SIGINT, &sig_action);
}
```

Пример Токио

Ящик tokio-signal обеспечивает решение для обработки сигналов, основанное на tokio. Тем не менее, он все еще находится на ранних стадиях.

```
extern crate futures;
extern crate tokio_core;
extern crate tokio_signal;

use futures::{Future, Stream};
use tokio_core::reactor::Core
use tokio_signal::unix::{self as unix_signal, Signal};
use std::thread::{self, sleep};
use std::time::Duration;
use std::sync::mpsc::{channel, Receiver};

fn run(signals: Receiver<i32>) {
    loop {
        if let Some(signal) = signals.try_recv() {
            eprintln!("received {} signal");
        }
        sleep(Duration::from_millis(1));
    }
}

fn main() {
    // Create channels for sending and receiving signals
    let (signals_tx, signals_rx) = channel();
```

```
    // Execute the program with the receiving end of the channel
    // for listening to any signals that are sent to it.
    thread::spawn(move || run(signals_rx));

    // Create a stream that will select over SIGINT, SIGTERM, and SIGHUP signals.
    let signals = Signal::new(unix_signal::SIGINT, &handle).flatten_stream()
        .select(Signal::new(unix_signal::SIGTERM, &handle).flatten_stream())
        .select(Signal::new(unix_signal::SIGHUP, &handle).flatten_stream());

    // Execute the event loop that will listen for and transmit received
    // signals to the shell.
    core.run(signals.for_each(|signal| {
        let _ = signals_tx.send(signal);
        Ok(())
    })).unwrap();
}
```

Прочитайте Обработка сигналов онлайн: https://riptutorial.com/ru/rust/topic/3995/обработка-сигналов

глава 31: Образцы конверсии

замечания

- `AsRef` и `Borrow` аналогичны, но служат различным целям. `Borrow` используется для обработки нескольких методов заимствования аналогичным образом или для обработки заемных ценностей, таких как принадлежащие им коллеги, в то время как `AsRef` используется для обобщения ссылок.
- `From<A> for B` подразумевается `Into<B> for A` , но не наоборот.
- `From<A> for A` неявно реализовано.

Examples

От

Rust's `From` trait - это универсальная черта для преобразования типов. Для любых двух типов `TypeA` и `TypeB` ,

```
impl From<TypeA> for TypeB
```

указывает, что экземпляр `TypeB` *гарантированно* будет создан из экземпляра `TypeA` . Реализация `From` выглядит следующим образом:

```
struct TypeA {
    a: u32,
}

struct TypeB {
    b: u32,
}

impl From<TypeA> for TypeB {
    fn from(src: TypeA) -> Self {
        TypeB {
            b: src.a,
        }
    }
}
```

AsRef & AsMut

`std::convert::AsRef` и `std::convert::AsMut` используются для дешевого преобразования типов в ссылки. Для типов `A` и `B` ,

```
impl AsRef<B> for A
```

указывает, что а `&A` может быть преобразовано в а `&B` и,

```
impl AsMut<B> for A
```

указывает, что а `&mut A` может быть преобразован в а `&mut B`

Это полезно для выполнения преобразований типов без копирования или перемещения значений. Пример в стандартной библиотеке: `std::fs::File.open()` :

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

Это позволяет `File.open()` принимать не только `Path` , но также `OsStr` , `OsString` , `str` , `String` и `PathBuf` с неявным преобразованием, поскольку эти типы реализуют `AsRef<Path>` .

Заимствовать, BorrowMut и ToOwned

Значения `std::borrow::Borrow` и `std::borrow::BorrowMut` используются для обработки заимствованных типов, таких как принадлежащие им типы. Для типов `A` и `B` ,

```
impl Borrow<B> for A
```

указывает, что заимствованный `A` может использоваться там, где требуется `B` Например, `std::collections::HashMap.get()` использует `Borrow` для метода `get()` , позволяя индексировать `HashMap` с ключами `A` с помощью `&B`

С другой стороны, `std::borrow::ToOwned` реализует обратную связь.

Таким образом, с вышеупомянутыми типами `A` и `B` можно реализовать:

```
impl ToOwned for B
```

Примечание: в то время как `A` может реализовать `Borrow<T>` для нескольких разных типов `T` , `B` может реализовать `ToOwned` однократно `ToOwned` .

Deref & DerefMut

`std::ops::Deref` и `std::ops::DerefMut` используются для перегрузки оператора разыменования `*x` . Для типов `A` и `B` ,

```
impl Deref<Target=B> for A
```

указывает, что разыменование связывания `&A` даст а `&B` и,

```
impl DerefMut for A
```

указывает на то, что разыменование связывания `&mut A` даст а `&mut B`

`Deref` (соответственно `DerefMut`) также предоставляет полезную функцию языка, называемую *deref coercion* , которая позволяет `&A` (resp. `&mut A`) автоматически принуждать к `&B` (соответственно `&mut B`). Это обычно используется при преобразовании из `String` в `&str` , поскольку `&String` неявно принудительно привязывается к `&str` мере необходимости.

Примечание. `DerefMut` не поддерживает указание результирующего типа, он использует тот же тип, что и `Deref` .

Примечание. Из-за использования связанного типа (в отличие от `AsRef`) данный тип может выполнять только каждый из `Deref` и `DerefMut` не более одного раза.

Прочитайте Образцы конверсии онлайн: https://riptutorial.com/ru/rust/topic/2661/образцы-конверсии

глава 32: Объектно-ориентированная ржавчина

Вступление

Rust объектно ориентирован на то, что его алгебраические типы данных могут иметь связанные методы, делая их объектами в смысле данных, хранящихся вместе с кодом, который знает, как с ним работать.

Однако ржавчина не поддерживает наследование, предпочитая композицию с чертами. Это означает, что многие шаблоны OO не работают как есть и должны быть изменены. Некоторые из них совершенно неактуальны.

Examples

Наследование с помощью свойств

В Rust нет понятия «наследовать» свойства структуры. Вместо этого, когда вы разрабатываете взаимосвязь между объектами, делайте это так, чтобы функциональность определялась интерфейсом (**черта** в Rust). Это способствует составлению над наследованием , которое считается более полезным и более простым для более крупных проектов.

Вот пример использования некоторого примера наследования в Python:

```
class Animal:
    def speak(self):
        print("The " + self.animal_type + " said " + self.noise)

class Dog(Animal):
    def __init__(self):
        self.animal_type = 'dog'
        self.noise = 'woof'
```

Чтобы перевести это в Rust, нам нужно вынуть то, что составляет животное, и превратить эту функциональность в черты.

```
trait Speaks {
     fn speak(&self);

     fn noise(&self) -> &str;
}

trait Animal {
    fn animal_type(&self) -> &str;
}
```

```
struct Dog {}

impl Animal for Dog {
    fn animal_type(&self) -> &str {
        "dog"
    }
}

impl Speaks for Dog {
    fn speak(&self) {
        println!("The dog said {}", self.noise());
    }

    fn noise(&self) -> &str {
        "woof"
    }
}

fn main() {
    let dog = Dog {};
    dog.speak();
}
```

Обратите внимание, как мы разбили этот абстрактный родительский класс на два отдельных компонента: часть, которая определяет структуру как животное, и ту часть, которая позволяет ему говорить.

Астуальные читатели заметят, что это не совсем одно к одному, так как каждый разработчик должен переделать логику, чтобы распечатать строку в форме «{animal} said {noise}». Вы можете сделать это с небольшой переделкой интерфейса, где мы реализуем `Speak` for `Animal` :

```
trait Speaks {
     fn speak(&self);
}

trait Animal {
    fn animal_type(&self) -> &str;
    fn noise(&self) -> &str;
}

impl<T> Speaks for T where T: Animal {
    fn speak(&self) {
        println!("The {} said {}", self.animal_type(), self.noise());
    }
}

struct Dog {}
struct Cat {}

impl Animal for Dog {
    fn animal_type(&self) -> &str {
        "dog"
    }

    fn noise(&self) -> &str {
```

```
        "woof"
    }
}

impl Animal for Cat {
    fn animal_type(&self) -> &str {
        "cat"
    }

    fn noise(&self) -> &str {
        "meow"
    }
}

fn main() {
    let dog = Dog {};
    let cat = Cat {};
    dog.speak();
    cat.speak();
}
```

Обратите внимание, что животное делает шум и говорит просто сейчас имеет реализацию для всего, что является животным. Это намного более гибко, чем предыдущий способ и наследование Python. Например, если вы хотите добавить `Human` , у которого есть другой звук, вместо этого у нас может быть другая реализация `speak` для чего-то `Human` :

```
trait Human {
    fn name(&self) -> &str;
    fn sentence(&self) -> &str;
}

struct Person {}

impl<T> Speaks for T where T: Human {
    fn speak(&self) {
        println!("{} said {}", self.name(), self.sentence());
    }
}
```

Шаблон посетителя

Типичным примером посетителя в Java будет:

```
interface ShapeVisitor {
    void visit(Circle c);
    void visit(Rectangle r);
}

interface Shape {
    void accept(ShapeVisitor sv);
}

class Circle implements Shape {
    private Point center;
    private double radius;
```

```
    public Circle(Point center, double radius) {
        this.center = center;
        this.radius = radius;
    }

    public Point getCenter() { return center; }
    public double getRadius() { return radius; }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class Rectangle implements Shape {
    private Point lowerLeftCorner;
    private Point upperRightCorner;

    public Rectangle(Point lowerLeftCorner, Point upperRightCorner) {
        this.lowerLeftCorner = lowerLeftCorner;
        this.upperRightCorner = upperRightCorner;
    }

    public double length() { ... }
    public double width() { ... }

    @Override
    public void accept(ShapeVisitor sv) {
        sv.visit(this);
    }
}

class AreaCalculator implements ShapeVisitor {
    private double area = 0.0;

    public double getArea() { return area; }

    public void visit(Circle c) {
        area = Math.PI * c.radius() * c.radius();
    }

    public void visit(Rectangle r) {
         area = r.length() * r.width();
    }
}

double computeArea(Shape s) {
    AreaCalculator ac = new AreaCalculator();
    s.accept(ac);
    return ac.getArea();
}
```

Это можно легко перевести на Rust двумя способами.

Первый способ использует полиморфизм во время выполнения:

```
trait ShapeVisitor {
    fn visit_circle(&mut self, c: &Circle);
    fn visit_rectangle(&mut self, r: &Rectangle);
```

```
}

trait Shape {
    fn accept(&self, sv: &mut ShapeVisitor);
}

struct Circle {
    center: Point,
    radius: f64,
}

struct Rectangle {
    lowerLeftCorner: Point,
    upperRightCorner: Point,
}

impl Shape for Circle {
    fn accept(&self, sv: &mut ShapeVisitor) {
        sv.visit_circle(self);
    }
}

impl Rectangle {
    fn length() -> double { ... }
    fn width() -> double { ... }
}

impl Shape for Rectangle {
    fn accept(&self, sv: &mut ShapeVisitor) {
        sv.visit_rectangle(self);
    }
}

fn computeArea(s: &Shape) -> f64 {
    struct AreaCalculator {
        area: f64,
    }

    impl ShapeVisitor for AreaCalculator {
        fn visit_circle(&mut self, c: &Circle) {
            self.area = std::f64::consts::PI * c.radius * c.radius;
        }
        fn visit_rectangle(&mut self, r: &Rectangle) {
            self.area = r.length() * r.width();
        }
    }

    let mut ac = AreaCalculator { area: 0.0 };
    s.accept(&mut ac);
    ac.area
}
```

Второй способ использует полиморфизм времени компиляции, здесь показаны только различия:

```
trait Shape {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V);
}
```

```
impl Shape for Circle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        // same body
    }
}

impl Shape for Rectangle {
    fn accept<V: ShapeVisitor>(&self, sv: &mut V) {
        // same body
    }
}

fn computeArea<S: Shape>(s: &S) -> f64 {
    // same body
}
```

Прочитайте Объектно-ориентированная ржавчина онлайн:
https://riptutorial.com/ru/rust/topic/6737/объектно-ориентированная-ржавчина

глава 33: Операторы и перегрузка

Вступление

Большинство операторов в Rust могут быть определены («перегружены») для пользовательских типов. Это может быть достигнуто путем реализации соответствующего признака в модуле `std::ops` .

Examples

Перегрузка оператора сложения (+)

Для перегрузки оператора сложения (+) требуется реализовать `std::ops::Add` trait.

Из документации полное определение признака:

```
pub trait Add<RHS = Self> {
    type Output;
    fn add(self, rhs: RHS) -> Self::Output;
}
```

Как это работает?

- черта реализована для типа «Левая рука»
- признак реализуется для *одного* аргумента правой руки, если не указано, что он по умолчанию имеет тот же тип, что и левая сторона стороны
- тип результата добавления указан в соответствующем типе `Output`

Таким образом, возможны 3 разных типа.

Примечание: потребляемый признак - это аргументы с левой стороны и правой стороны, вы можете предпочесть реализовать его для ссылок на ваш тип, а не на голые типы.

Внедрение `+` для настраиваемого типа:

```
use std::ops::Add;

#[derive(Clone)]
struct List<T> {
    data: Vec<T>,
}
//  Implementation which consumes both LHS and RHS
impl<T> Add for List<T> {
    type Output = List<T>;

    fn add(self, rhs: List<T>) -> List<T> {
```

```
        self.data.extend(rhs.data.drain(..));
        self
    }
}

//  Implementation which only consumes RHS (and thus where LHS != RHS)
impl<'a, T: Clone> Add<List<T>> for &'a List<T> {
    type Output = List<T>;

    fn add(self, rhs: List<T>) -> List<T> {
        self.clone() + rhs
    }
}
```

Прочитайте Операторы и перегрузка онлайн: https://riptutorial.com/ru/rust/topic/7271/операторы-и-перегрузка

глава 34: Паники и разматывания

Вступление

Когда программы Rust достигают состояния, где произошла критическая ошибка, `panic!` макрос может вызываться для быстрого выхода (часто сравнивается, но тонко отличается от исключения на других языках). Правильная обработка ошибок должна включать в себя типы `Result` , хотя в этом разделе будет обсуждаться только `panic!` и его концепций.

замечания

Паники не всегда вызывают утечки памяти или утечки других ресурсов. На самом деле, паники обычно сохраняют инварианты RAII, запуская деструкторы (Drop-реализаций) структур при раскручивании стека. Однако, если во время этого процесса возникает вторая паника, программа просто прерывается; в этот момент инвариантные гарантии RAII являются недействительными.

Examples

Старайтесь не паниковать

В Rust есть два основных метода, указывающих на то, что в программе что-то пошло не так: функция, возвращающая (потенциально настраиваемый) `Err(E)` из типа `Result<T, E>` и `panic!` ,

Паника **не** является альтернативой исключениям, которые обычно встречаются на других языках. В Ржавчине паника указывает на то, что что-то пошло не так, и что она не может продолжаться. Вот пример из источника `Vec` для `push` :

```
pub fn push(&mut self, value: T) {
    // This will panic or abort if we would allocate > isize::MAX bytes
    // or if the length increment would overflow for zero-sized types.
    if self.len == self.buf.cap() {
        self.buf.double();
    }
    ...
}
```

Если у нас заканчивается память, Rust может не так много сделать, поэтому она будет либо паниковать (поведение по умолчанию), либо прерывать (что необходимо установить с помощью флага компиляции).

Паника будет разворачивать стек, запускать деструкторы и обеспечивать очистку памяти. Abort не делает этого и полагается на ОС для правильной очистки.

Попробуйте запустить следующую программу как обычно, так и с помощью

```
[profile.dev]
panic = "abort"
```

в вашем `Cargo.toml` .

```
// main.rs
struct Foo(i32);
impl Drop for Foo {
    fn drop(&mut self) {
        println!("Dropping {:?}!", self.0);
    }
}
fn main() {
    let foo = Foo(1);
    panic!("Aaaaaaahhhhh!");
}
```

Прочитайте Паники и разматывания онлайн: https://riptutorial.com/ru/rust/topic/6895/паники-и-разматывания

глава 35: параллелизм

Вступление

Параллелизм хорошо поддерживается стандартной библиотекой Rust через различные классы, такие как `std::thread` module, channels и atomics. В этом разделе вы узнаете об использовании этих типов.

Examples

Запуск новой темы

Чтобы начать новый поток:

```
use std::thread;

fn main() {
    thread::spawn(move || {
        // The main thread will not wait for this thread to finish. That
        // might mean that the next println isn't even executed before the
        // program exits.
        println!("Hello from spawned thread");
    });

    let join_handle = thread::spawn(move || {
        println!("Hello from second spawned thread");
        // To ensure that the program waits for a thread to finish, we must
        // call `join()` on its join handle. It is even possible to send a
        // value to a different thread through the join handle, like the
        // integer 17 in this case:
        17
    });

    println!("Hello from the main thread");

    // The above three printlns can be observed in any order.

    // Block until the second spawned thread has finished.
    match join_handle.join() {
        Ok(x) => println!("Second spawned thread returned {}", x),
        Err(_) => println!("Second spawned thread panicked")
    }
}
```

Поперечное взаимодействие с каналами

Каналы могут использоваться для передачи данных из одного потока в другой. Ниже приведен пример простой системы «производитель-потребитель», где основной поток создает значения 0, 1, ..., 9, а порожденный поток печатает их:

```
use std::thread;
use std::sync::mpsc::channel;

fn main() {
    // Create a channel with a sending end (tx) and a receiving end (rx).
    let (tx, rx) = channel();

    // Spawn a new thread, and move the receiving end into the thread.
    let join_handle = thread::spawn(move || {
        // Keep receiving in a loop, until tx is dropped!
        while let Ok(n) = rx.recv() { // Note: `recv()` always blocks
            println!("Received {}", n);
        }
    });

    // Note: using `rx` here would be a compile error, as it has been
    // moved into the spawned thread.

    // Send some values to the spawned thread. `unwrap()` crashes only if the
    // receiving end was dropped before it could be buffered.
    for i in 0..10 {
        tx.send(i).unwrap(); // Note: `send()` never blocks
    }

    // Drop `tx` so that `rx.recv()` returns an `Err(_)`.
    drop(tx);

    // Wait for the spawned thread to finish.
    join_handle.join().unwrap();
}
```

Межпоточная связь с типами сеансов

Типы сеансов - это способ рассказать компилятору о протоколе, который вы хотите использовать для связи между потоками, а не протоколе, как в HTTP или FTP, а о шаблоне потока информации между потоками. Это полезно, так как компилятор теперь остановит вас от случайного нарушения вашего протокола и возникновения взаимоблокировок или живого потока между потоками - некоторые из наиболее печально известных проблем с отладкой и главный источник Heisenbugs. Типы сеансов работают аналогично описанным выше каналам, но могут быть более запугивающими для начала использования. Вот простая двухпоточная связь:

```
// Session Types aren't part of the standard library, but are part of this crate.
// You'll need to add session_types to your Cargo.toml file.
extern crate session_types;

// For now, it's easiest to just import everything from the library.
use session_types::*;

// First, we describe what our client thread will do. Note that there's no reason
// you have to use a client/server model - it's just convenient for this example.
// This type says that a client will first send a u32, then quit. `Eps` is
// shorthand for "end communication".
// Session Types use two generic parameters to describe the protocol - the first
// for the current communication, and the second for what will happen next.
type Client = Send<u32, Eps>;
```

```
// Now, we define what the server will do: it will receive as u32, then quit.
type Server = Recv<u32, Eps>;

// This function is ordinary code to run the client. Notice that it takes
// ownership of a channel, just like other forms of interthread communication -
// but this one about the protocol we just defined.
fn run_client(channel: Chan<(), Client>) {
    let channel = channel.send(42);
    println!("The client just sent the number 42!");
    channel.close();
}

// Now we define some code to run the server. It just accepts a value and prints
// it.
fn run_server(channel: Chan<(), Server>) {
    let (channel, data) = channel.recv();
    println!("The server received some data: {}", data);
    channel.close();
}

fn main() {
    // First, create the channels used for the two threads to talk to each other.
    let (server_channel, client_channel) = session_channel();

    // Start the server on a new thread
    let server_thread = std::thread::spawn(move || {
        run_server(server_channel);
    });

    // Run the client on this thread.
    run_client(client_channel);

    // Wait for the server to finish.
    server_thread.join().unwrap();
}
```

Вы должны заметить, что основной метод очень похож на основной метод межпоточной связи, определенный выше, если сервер был перемещен в свою собственную функцию. Если вы хотите запустить это, вы получите результат:

```
The client just sent the number 42!
The server received some data: 42
```

в этой последовательности.

Зачем решать все проблемы с определением типов клиентов и серверов? И почему мы переопределяем канал на клиенте и сервере? Эти вопросы имеют один и тот же ответ: компилятор остановит нас от нарушения протокола! Если клиент попытался получить данные вместо их отправки (что приведет к тупиковой ситуации в обычном коде), программа не будет компилироваться, поскольку объект канала клиента *не имеет на нем метода* `recv`. Кроме того, если мы попытались определить протокол таким образом, который может привести к тупиковой ситуации (например, если и клиент, и сервер попытались получить значение), тогда компиляция завершится неудачно, когда мы создадим каналы. Это связано с тем, что `Send` и `Recv` являются « `Recv` типами», то есть если

сервер делает это, клиент должен сделать другой - если оба попытаются `Recv` , у вас будут проблемы. `Eps` - это свой собственный двойной тип, так как для клиента и сервера вполне нормально соглашаться закрыть канал.

Конечно, когда мы выполняем некоторую операцию на канале, мы переходим к новому состоянию в протоколе, и доступные нам функции могут измениться - поэтому нам нужно переопределить привязку канала. К счастью, `session_types` позаботится об этом для нас и всегда возвращает новый канал (кроме `close` , и в этом случае нет нового канала). Это также означает, что все методы на канале также владеют каналом, поэтому, если вы забудете переопределить канал, компилятор также даст вам об этом ошибку. Если вы отпустите канал, не закрывая его, это также ошибка времени выполнения (к сожалению, это невозможно проверить во время компиляции).

Существует гораздо больше типов связи, чем просто `Send` and `Recv` - например, `Offer` дает другой стороне канала возможность выбирать между двумя возможными ветвями протокола, а `Rec` и `Var` работают вместе, чтобы разрешить циклы и рекурсию в протоколе , В репозитории `session_types` GitHub доступно еще много примеров типов сеансов и других типов. Документацию библиотеки можно найти здесь.

Атомная и память

Атомные типы являются строительными блоками незакрепленных структур данных и других параллельных типов. При доступе к / модификации атомного типа следует указывать порядок памяти, представляющий силу барьера памяти. Rust обеспечивает 5 примитивов упорядочения памяти: **Relaxed** (самый слабый), **Acquire** (для чтения aka load), **Release** (для записи aka магазинов), **AcqRel** (эквивалент «Приобретать для загрузки и выпуска для хранения»; полезно, когда оба участвуют в одной операции, такой как compare-and-swap) и **SeqCst** (самый сильный). В приведенном ниже примере мы продемонстрируем, как порядок «Relaxed» отличается от порядка «Приобретать» и «Отпускать».

```
use std::cell::UnsafeCell;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::{Arc, Barrier};
use std::thread;

struct UsizePair {
    atom: AtomicUsize,
    norm: UnsafeCell<usize>,
}

// UnsafeCell is not thread-safe. So manually mark our UsizePair to be Sync.
// (Effectively telling the compiler "I'll take care of it!")
unsafe impl Sync for UsizePair {}

static NTHREADS: usize = 8;
static NITERS: usize = 1000000;

fn main() {
```

```
    let upair = Arc::new(UsizePair::new(0));

    // Barrier is a counter-like synchronization structure (not to be confused
    // with a memory barrier). It blocks on a `wait` call until a fixed number
    // of `wait` calls are made from various threads (like waiting for all
    // players to get to the starting line before firing the starter pistol).
    let barrier = Arc::new(Barrier::new(NTHREADS + 1));

    let mut children = vec![];

    for _ in 0..NTHREADS {
        let upair = upair.clone();
        let barrier = barrier.clone();
        children.push(thread::spawn(move || {
            barrier.wait();

            let mut v = 0;
            while v < NITERS - 1 {
                // Read both members `atom` and `norm`, and check whether `atom`
                // contains a newer value than `norm`. See `UsizePair` impl for
                // details.
                let (atom, norm) = upair.get();
                if atom > norm {
                    // If `Acquire`-`Release` ordering is used in `get` and
                    // `set`, then this statement will never be reached.
                    println!("Reordered! {} > {}", atom, norm);
                }
                v = atom;
            }
        }));
    }

    barrier.wait();

    for v in 1..NITERS {
        // Update both members `atom` and `norm` to value `v`. See the impl for
        // details.
        upair.set(v);
    }

    for child in children {
        let _ = child.join();
    }
}

impl UsizePair {
    pub fn new(v: usize) -> UsizePair {
        UsizePair {
            atom: AtomicUsize::new(v),
            norm: UnsafeCell::new(v),
        }
    }

    pub fn get(&self) -> (usize, usize) {
        let atom = self.atom.load(Ordering::Relaxed); //Ordering::Acquire

        // If the above load operation is performed with `Acquire` ordering,
        // then all writes before the corresponding `Release` store is
        // guaranteed to be visible below.

        let norm = unsafe { *self.norm.get() };
```

```
        (atom, norm)
    }

    pub fn set(&self, v: usize) {
        unsafe { *self.norm.get() = v };

        // If the below store operation is performed with `Release` ordering,
        // then the write to `norm` above is guaranteed to be visible to all
        // threads that "loads `atom` with `Acquire` ordering and sees the same
        // value that was stored below". However, no guarantees are provided as
        // to when other readers will witness the below store, and consequently
        // the above write. On the other hand, there is also no guarantee that
        // these two values will be in sync for readers. Even if another thread
        // sees the same value that was stored below, it may actually see a
        // "later" value in `norm` than what was written above. That is, there
        // is no restriction on visibility into the future.

        self.atom.store(v, Ordering::Relaxed); //Ordering::Release
    }
}
```

Примечание. Архитектуры x86 имеют сильную модель памяти. Эта статья объясняет это подробно. Также взгляните на страницу Википедии для сравнения архитектур.

Блокировки чтения и записи

RwLocks позволяют одному производителю предоставлять любое количество считывателей с данными, не допуская, чтобы читатели видели недопустимые или несогласованные данные.

В следующем примере RwLock показывает, как один поток производителей может периодически увеличивать значение, а два потока потребителей считывают значение.

```
use std::time::Duration;
use std::thread;
use std::thread::sleep;
use std::sync::{Arc, RwLock };

fn main() {
    // Create an u32 with an inital value of 0
    let initial_value = 0u32;

    // Move the initial value into the read-write lock which is wrapped into an atomic
reference
    // counter in order to allow safe sharing.
    let rw_lock = Arc::new(RwLock::new(initial_value));

    // Create a clone for each thread
    let producer_lock = rw_lock.clone();
    let consumer_id_lock = rw_lock.clone();
    let consumer_square_lock = rw_lock.clone();

    let producer_thread = thread::spawn(move || {
        loop {
            // write() blocks this thread until write-exclusive access can be acquired and
retuns an
```

```
            // RAII guard upon completion
            if let Ok(mut write_guard) = producer_lock.write() {
                // the returned write_guard implements `Deref` giving us easy access to the
target value
                *write_guard += 1;

                println!("Updated value: {}", *write_guard);
            }

            // ^
            // |   when the RAII guard goes out of the scope, write access will be dropped,
allowing
            // +~  other threads access the lock

            sleep(Duration::from_millis(1000));
        }
    });

    // A reader thread that prints the current value to the screen
    let consumer_id_thread = thread::spawn(move || {
        loop {
            // read() will only block when `producer_thread` is holding a write lock
            if let Ok(read_guard) = consumer_id_lock.read() {
                // the returned read_guard also implements `Deref`
                println!("Read value: {}", *read_guard);
            }

            sleep(Duration::from_millis(500));
        }
    });

    // A second reader thread is printing the squared value to the screen. Note that readers
don't
    // block each other so `consumer_square_thread` can run simultaneously with
`consumer_id_lock`.
    let consumer_square_thread = thread::spawn(move || {
        loop {
            if let Ok(lock) = consumer_square_lock.read() {
                let value = *lock;
                println!("Read value squared: {}", value * value);
            }

            sleep(Duration::from_millis(750));
        }
    });

    let _ = producer_thread.join();
    let _ = consumer_id_thread.join();
    let _ = consumer_square_thread.join();
}
```

Пример вывода:

```
Updated value: 1
Read value: 1
Read value squared: 1
Read value: 1
Read value squared: 1
Updated value: 2
Read value: 2
```

```
Read value: 2
Read value squared: 4
Updated value: 3
Read value: 3
Read value squared: 9
Read value: 3
Updated value: 4
Read value: 4
Read value squared: 16
Read value: 4
Read value squared: 16
Updated value: 5
Read value: 5
Read value: 5
Read value squared: 25
...(Interrupted)...
```

Прочитайте параллелизм онлайн: https://riptutorial.com/ru/rust/topic/1222/параллелизм

глава 36: Пользовательский вывод: «Макросы 1.1»

Вступление

Rust 1.15 добавила (стабилизировала) новую функцию: Custom вывести aka Macros 1.1.

Теперь, помимо обычных `PartialEq` или `Debug` вы можете получить `#[deriving(MyOwnDerive)]` . Двумя основными пользователями этой функции являются серд и дизель .

Rust Ссылка на книгу: https://doc.rust-lang.org/stable/book/procedural-macros.html

Examples

Вербальный неряшливый helloworld

Cargo.toml:

```
[package]
name = "customderive"
version = "0.1.1"

[lib]
proc-macro=true

[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

SRC / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
#[macro_use]
extern crate quote;

#[proc_macro_derive(Hello)]
pub fn qqq(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    println!("Normalized source code: {}", source);
    let ast = syn::parse_derive_input(&source).unwrap();
    println!("Syn's AST: {:?}", ast); // {:#?} - pretty print
    let struct_name = &ast.ident;
    let quoted_code = quote!{
        fn hello() {
            println!("Hello, {}!", stringify!(#struct_name));
```

```
        }
    };
    println!("Quoted code: {:?}", quoted_code);
    quoted_code.parse().unwrap()
}
```

Примеры / hello.rs:

```
#[macro_use]
extern crate customderive;

#[derive(Hello)]
struct Qqq;

fn main(){
    hello();
}
```

выход:

```
$ cargo run --example hello
   Compiling customderive v0.1.1 (file:///tmp/cd)
Normalized source code: struct Qqq;
Syn's AST: DeriveInput { ident: Ident("Qqq"), vis: Inherited, attrs: [], generics: Generics {
lifetimes: [], ty_params: [], where_clause: WhereClause { predicates: [] } }, body:
Struct(Unit) }
Quoted code: Tokens("fn hello ( ) { println ! ( \"Hello, {}!\" , stringify ! ( Qqq ) ) ; }")
warning: struct is never used: <snip>
    Finished dev [unoptimized + debuginfo] target(s) in 3.79 secs
     Running `target/x86_64-unknown-linux-gnu/debug/examples/hello`
Hello, Qqq!
```

Минимальный фиктивный пользовательский вывод

Cargo.toml:

```
[package]
name = "customderive"
version = "0.1.0"
[lib]
proc-macro=true
```

SRC / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(Dummy)]
pub fn qqq(input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

примеры / hello.rs

```
#[macro_use]
extern crate customderive;

#[derive(Dummy)]
struct Qqq;

fn main(){}
```

Геттеры и сеттеры

Cargo.toml:

```
[package]
name = "gettersetter"
version = "0.1.0"
[lib]
proc-macro=true
[dependencies]
quote="^0.3.12"
syn="^0.11.4"
```

SRC / lib.rs:

```
#![crate_type = "proc-macro"]
extern crate proc_macro;
use proc_macro::TokenStream;

extern crate syn;
#[macro_use]
extern crate quote;

#[proc_macro_derive(GetSet)]
pub fn qqq(input: TokenStream) -> TokenStream {
    let source = input.to_string();
    let ast = syn::parse_derive_input(&source).unwrap();

    let struct_name = &ast.ident;
    if let syn::Body::Struct(s) = ast.body {
        let field_names : Vec<_> = s.fields().iter().map(|ref x|
                x.ident.clone().unwrap()).collect();

        let field_getter_names = field_names.iter().map(|ref x|
                syn::Ident::new(format!("get_{}", x).as_str()));
        let field_setter_names = field_names.iter().map(|ref x|
                syn::Ident::new(format!("set_{}", x).as_str()));
        let field_types : Vec<_> = s.fields().iter().map(|ref x|
                x.ty.clone()).collect();
        let field_names2 = field_names.clone();
        let field_names3 = field_names.clone();
        let field_types2 = field_types.clone();

        let quoted_code = quote!{
            #[allow(dead_code)]
            impl #struct_name {
                #(
                    fn #field_getter_names(&self) -> &#field_types {
                        &self.#field_names2
                    }
```

```
                    fn #field_setter_names(&mut self, x : #field_types2) {
                        self.#field_names3 = x;
                    }
                )*
            }
        };
        return quoted_code.parse().unwrap();
    }
    // not a struct
    "".parse().unwrap()
}
```

Примеры / hello.rs:

```
#[macro_use]
extern crate gettersetter;

#[derive(GetSet)]
struct Qqq {
    x : i32,
    y : String,
}

fn main(){
    let mut a = Qqq { x: 3, y: "zzaaqq".to_string() };
    println!("{}", a.get_x());
    a.set_y("123213".to_string());
    println!("{}", a.get_y());
}
```

См. Также: https://github.com/emk/accessors

Прочитайте Пользовательский вывод: «Макросы 1.1» онлайн: https://riptutorial.com/ru/rust/topic/9104/пользовательский-вывод---макросы-1-1-

глава 37: Приложения для графического интерфейса пользователя

Вступление

У Rust нет собственной основы для разработки GUI. Тем не менее, существует множество привязок к существующим структурам. Самым передовым связыванием библиотеки является rust-gtk . Полный список переплетов можно найти здесь

Examples

Простое окно Gtk + с текстом

Добавьте Gtk dependecy в свой `Cargo.toml` :

```
[dependencies]
gtk = { git = "https://github.com/gtk-rs/gtk.git" }
```

Создайте простое окно со следующим:

```
extern crate gtk;

use gtk::prelude::*; // Import all the basic things
use gtk::{Window, WindowType, Label};

fn main() {
    if gtk::init().is_err() { //Initialize Gtk before doing anything with it
        panic!("Can't init GTK");
    }

    let window = Window::new(WindowType::Toplevel);

    //Destroy window on exit
    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));
    window.add(&label);
    window.show_all();
    gtk::main();
}
```

Окно Gtk + со входом и меткой в GtkBox, соединение GtkEntry

```
extern crate gtk;
```

```
use gtk::prelude::*;
use gtk::{Window, WindowType, Label, Entry, Box as GtkBox, Orientation};

fn main() {
    if gtk::init().is_err() {
        println!("Failed to initialize GTK.");
        return;
    }

    let window = Window::new(WindowType::Toplevel);

    window.connect_delete_event(|_,_| {gtk::main_quit(); Inhibit(false) });

    window.set_title("Stackoverflow. example");
    window.set_default_size(350, 70);
    let label = Label::new(Some("Some text"));

    // Create a VBox with 10px spacing
    let bx = GtkBox::new(Orientation::Vertical, 10);
    let entry = Entry::new();

    // Connect "activate" signal to anonymous function
    // that takes GtkEntry as an argument and prints it's text
    entry.connect_activate(|x| println!("{}",x.get_text().unwrap()));

    // Add our label and entry to the box
    // Do not expand or fill, zero padding
    bx.pack_start(&label, false, false, 0);
    bx.pack_start(&entry, false, false, 0);
    window.add(&bx);
    window.show_all();
    gtk::main();
}
```

Прочитайте Приложения для графического интерфейса пользователя онлайн: https://riptutorial.com/ru/rust/topic/7169/приложения-для-графического-интерфейса-пользователя

глава 38: Примитивные типы данных

Examples

Скалярные типы

Целые

Подпись : `i8` , `i16` , `i32` , `i64` , `isize`

Unsigned : `u8` , `u16` , `u32` , `u64` , `usize`

Тип целочисленного литерала, скажем `45` , будет автоматически выводиться из контекста. Но чтобы заставить его добавить суффикс: `45u8` (без пробела) будет напечатан `u8` .

Примечание. Размер `isize` и `usize` зависит от архитектуры. На 32-битной архитектуре это 32-битные, а на 64-битных, вы догадались!

Плавающие точки

`f32` и `f64` .

Если вы просто пишете `2.0` , это по умолчанию `f64` , если только вывод типа не определяет иначе!

Чтобы заставить `f32` , либо определить переменную с `f32` типа, или суффикс литерала: `2.0f32` .

Булевы

`bool` , имеющий значения `true` и `false` .

Персонажи

`char` , со значениями, записанными как `'x'` . В одинарных кавычках содержится одно значение Scicar Value Unicode, что означает, что в нем есть эмулятор! Вот еще 3 примера: `''` , `'\u{3f}'` , `'\u{1d160}'` .

Прочитайте Примитивные типы данных онлайн: https://riptutorial.com/ru/rust/topic/8705/

примитивные-типы-данных

глава 39: Руководство по стилю ржавчины

Вступление

Несмотря на отсутствие официального руководства по стилю Rust, следующие примеры показывают соглашения, принятые большинством проектов Rust. Следуя этим соглашениям, вы сравните стиль вашего проекта со стилем стандартной библиотеки, что упростит для людей возможность увидеть логику вашего кода.

замечания

Официальные правила стиля Руста были доступны в репозитории `rust-lang/rust` на GitHub, но они были недавно удалены, до перехода в репозиторий `rust-lang-nursery/fmt-rfcs` . Пока новые руководящие принципы не будут опубликованы там, вы должны попытаться следовать рекомендациям в репозитории `rust-lang` .

Вы можете использовать rustfmt и clippy для автоматического просмотра кода для проблем стиля и правильного его форматирования. Эти инструменты могут быть установлены с использованием Cargo, например:

```
cargo install clippy
cargo install rustfmt
```

Чтобы запустить их, вы используете:

```
cargo clippy
cargo fmt
```

Examples

Пробелы

Длина линии

```
// Lines should never exceed 100 characters.
// Instead, just wrap on to the next line.
let bad_example = "So this sort of code should never really happen, because it's really hard
to fit on the screen!";
```

вдавливание

```
// You should always use 4 spaces for indentation.
// Tabs are discouraged - if you can, set your editor to convert
// a tab into 4 spaces.
```

```
let x = vec![1, 3, 5, 6, 7, 9];
for item in x {
    if x / 2 == 3 {
        println!("{}", x);
    }
}
```

Прокручивание пробелов

Замыкание пробелов в конце файлов или строк должно быть удалено.

Двоичные операторы

```
// For clarity, always add a space when using binary operators, e.g.
// +, -, =, *
let bad=3+4;
let good = 3 + 4;
```

Это также относится к атрибутам, например:

```
// Good:
#[deprecated = "Don't use my class - use Bar instead!"]

// Bad:
#[deprecated="This is broken"]
```

Точка с запятой

```
// There is no space between the end of a statement
// and a semicolon.

let bad = Some("don't do this!") ;
let good: Option<&str> = None;
```

Выравнивание полей структуры

```
// Struct fields should **not** be aligned using spaces, like this:
pub struct Wrong {
    pub x  : i32,
    pub foo: i64
}

// Instead, just leave 1 space after the colon and write the type, like this:
pub struct Right {
    pub x: i32,
    pub foo: i64
}
```

Функциональные подписи

```
// Long function signatures should be wrapped and aligned so that
// the starting parameter of each line is aligned
fn foo(example_item: Bar, another_long_example: Baz,
```

```
       yet_another_parameter: Quux)
       -> ReallyLongReturnItem {
    // Be careful to indent the inside block correctly!
}
```

фигурные скобки

```
// The starting brace should always be on the same line as its parent.
// The ending brace should be on its own line.
fn bad()
{
    println!("This is incorrect.");
}

struct Good {
    example: i32
}

struct AlsoBad {
    example: i32 }
```

Создание ящиков

Прелюдии и реэкспорт

```
// To reduce the amount of imports that users need, you should
// re-export important structs and traits.
pub use foo::Client;
pub use bar::Server;
```

Иногда ящики используют модуль `prelude` для хранения важных структур, как `std::io::prelude` . Обычно они импортируются с `use std::io::prelude::*;`

импорт

Вы должны заказать свой импорт и декларации следующим образом:

- объявления `extern crate`
- `use` импорт
    - Внешний импорт из других ящиков должен быть первым
- Реэкспорт (`pub use`)

Именование

Структуры

```
// Structs use UpperCamelCase.
pub struct Snafucator {

}
```

```
mod snafucators {
    // Try to avoid 'stuttering' by repeating
    // the module name in the struct name.

    // Bad:
    pub struct OrderedSnafucator {

    }

    // Good:
    pub struct Ordered {

    }
}
```

Черты

```
// Traits use the same naming principles as
// structs (UpperCamelCase).
trait Read {
    fn read_to_snafucator(&self) -> Result<(), Error>;
}
```

Ящики и модули

```
// Modules and crates should both use snake_case.
// Crates should try to use single words if possible.
extern crate foo;
mod bar_baz {
    mod quux {

    }
}
```

Статические переменные и константы

```
// Statics and constants use SCREAMING_SNAKE_CASE.
const NAME: &'static str = "SCREAMING_SNAKE_CASE";
```

Перечисления

```
// Enum types and their variants **both** use UpperCamelCase.
pub enum Option<T> {
   Some(T),
   None
}
```

Функции и методы

```
// Functions and methods use snake_case
fn snake_cased_function() {

}
```

Переменные привязки

```
// Regular variables also use snake_case
let foo_bar = "snafu";
```

Время жизни

```
// Lifetimes should consist of a single lower case letter. By
// convention, you should start at 'a, then 'b, etc.

// Good:
struct Foobar<'a> {
    x: &'a str
}

// Bad:
struct Bazquux<'stringlife> {
    my_str: &'stringlife str
}
```

Сокращения

Имена переменных, содержащие аббревиатуры, такие как `TCP` должны быть написаны следующим образом:

- Для имен `UpperCamelCase` **первая буква** должна быть заглавной (например, `TcpClient`)
- Для имен `snake_case` не должно быть капитализации (например, `tcp_client`)
- Для имен `SCREAMING_SNAKE_CASE` аббревиатура должна быть полностью заглавной (например, `TCP_CLIENT`)

Типы

Тип аннотации

```
// There should be one space after the colon of the type
// annotation. This rule applies in variable declarations,
// struct fields, functions and methods.

// GOOD:
let mut buffer: String = String::new();
// BAD:
let mut buffer:String = String::new();
let mut buffer : String = String::new();
```

Рекомендации

```
// The ampersand (&) of a reference should be 'touching'
// the type it refers to.

// GOOD:
let x: &str = "Hello, world.";
// BAD:
```

```
fn fooify(x: & str) {
    println!("{}", x);
}
```

```
// Mutable references should be formatted like so:
fn bar(buf: &mut String) {

}
```

Прочитайте Руководство по стилю ржавчины онлайн: https://riptutorial.com/ru/rust/topic/4620/руководство-по-стилю-ржавчины

глава 40: Связанные константы

Синтаксис

- #! [Особенность (associated_consts)]
- const ID: i32;

замечания

Эта функция доступна только в ночном компиляторе. Проблема отслеживания # 29646

Examples

Использование связанных констант

```
// Must enable the feature to use associated constants
#![feature(associated_consts)]

use std::mem;

// Associated constants can be used to add constant attributes to types
trait Foo {
    const ID: i32;
}

// All implementations of Foo must define associated constants
// unless a default value is supplied in the definition.
impl Foo for i32 {
    const ID: i32 = 1;
}

struct Bar;

// Associated constants don't have to be bound to a trait to be defined
impl Bar {
    const BAZ: u32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);

    // The defined constant value is only stored once, so the size of
    // instances of the defined types doesn't include the constants.
    assert_eq!(4, mem::size_of::<i32>());
    assert_eq!(0, mem::size_of::<Bar>());
}
```

Прочитайте Связанные константы онлайн: https://riptutorial.com/ru/rust/topic/7042/связанные-константы

глава 41: Сеть TCP

Examples

Простое клиентское и серверное приложение TCP: echo

Следующий код основан на примерах, представленных в документации по std :: net :: TcpListener . Это серверное приложение будет прослушивать входящие запросы и отправлять обратно все входящие данные, действуя как сервер «эхо». Клиентское приложение отправит небольшое сообщение и ожидает ответа с тем же содержимым.

сервер:

```
use std::thread;
use std::net::{TcpListener, TcpStream, Shutdown};
use std::io::{Read, Write};

fn handle_client(mut stream: TcpStream) {
    let mut data = [0 as u8; 50]; // using 50 byte buffer
    while match stream.read(&mut data) {
        Ok(size) => {
            // echo everything!
            stream.write(&data[0..size]).unwrap();
            true
        },
        Err(_) => {
            println!("An error occurred, terminating connection with {}",
stream.peer_addr().unwrap());
            stream.shutdown(Shutdown::Both).unwrap();
            false
        }
    } {}
}

fn main() {
    let listener = TcpListener::bind("0.0.0.0:3333").unwrap();
    // accept connections and process them, spawning a new thread for each one
    println!("Server listening on port 3333");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                println!("New connection: {}", stream.peer_addr().unwrap());
                thread::spawn(move|| {
                    // connection succeeded
                    handle_client(stream)
                });
            }
            Err(e) => {
                println!("Error: {}", e);
                /* connection failed */
            }
        }
    }
    // close the socket server
```

```
    drop(listener);
}
```

клиент:

```
use std::net::{TcpStream};
use std::io::{Read, Write};
use std::str::from_utf8;

fn main() {
    match TcpStream::connect("localhost:3333") {
        Ok(mut stream) => {
            println!("Successfully connected to server in port 3333");

            let msg = b"Hello!";

            stream.write(msg).unwrap();
            println!("Sent Hello, awaiting reply...");

            let mut data = [0 as u8; 6]; // using 6 byte buffer
            match stream.read_exact(&mut data) {
                Ok(_) => {
                    if &data == msg {
                        println!("Reply is ok!");
                    } else {
                        let text = from_utf8(&data).unwrap();
                        println!("Unexpected reply: {}", text);
                    }
                },
                Err(e) => {
                    println!("Failed to receive data: {}", e);
                }
            }
        },
        Err(e) => {
            println!("Failed to connect: {}", e);
        }
    }
    println!("Terminated.");
}
```

Прочитайте Сеть TCP онлайн: https://riptutorial.com/ru/rust/topic/1350/сеть-tcp

глава 42: сооружения

Синтаксис

- struct Foo {field1: Type1, field2: Type2}
- let foo = Foo {field1: Type1 :: new (), field2: Type2 :: new ()};
- struct Bar (Тип1, Тип2); // тип tuple
- let _ = Bar (Type1 :: new (), Type2 :: new ());
- структура Баз; // Единичный тип
- let _ = Baz;
- пусть Foo {field1, ...} = foo; // извлекаем поле1 по образцу
- пусть Foo {field1: x, ..} = foo; // извлекаем поле1 как x
- let foo2 = Foo {field1: Type1 :: new (), .. foo}; // строить из существующих
- impl Foo {fn fiddle (& self) {}} // объявлять метод экземпляра для Foo
- impl Foo {fn tweak (& mut self) {}} // объявлять метод изменяемого экземпляра для Foo
- impl Foo {fn double (self) {}} // объявлять метод экземпляра экземпляра для Foo
- impl Foo {fn new () {}} // объявлять связанный метод для Foo

Examples

Определение структур

Структуры в Rust определяются с помощью ключевого слова `struct` . Наиболее распространенная форма структуры состоит из набора именованных полей:

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
```

Вышеупомянутое объявляет `struct` с тремя полями: `my_bool` , `my_num` и `my_string` , типа `bool` , `isize` и `String` соответственно.

Другой способ создания `struct` s в Rust - создать *структуру кортежей* :

```
struct Bar (bool, isize, String);
```

Это определяет новый тип, `Bar` , который имеет три неназванных поля типа `bool` , `isize` и `String` в этом порядке. Это называется шаблоном *newtype* , потому что он эффективно вводит новое «имя» для определенного типа. Однако он делает это более мощным образом, чем псевдонимы, созданные с использованием ключевого слова `type` ; `Bar` здесь полностью функциональный, то есть вы можете написать для него свои собственные

методы (см. Ниже).

Наконец, объявите `struct` *без* полей, называемую *структурно подобранной структурой* :

```
struct Baz;
```

Это может быть полезно для насмешек или тестирования (когда вы хотите тривиально реализовать признак) или как тип маркера. В целом, однако, вы вряд ли столкнетесь со многими структурными единицами.

Обратите внимание: поля `struct` в Rust являются закрытыми по умолчанию, то есть они не могут быть доступны из кода вне модуля, который определяет тип. Вы можете префикс поля с ключевым словом `pub` чтобы сделать это поле общедоступным. Кроме того, `struct` *сама* по *себе* тип является частным. Чтобы сделать тип доступным для других модулей, определение `struct` также должно иметь префикс `pub` :

```
pub struct X {
    my_field: bool,
    pub our_field: bool,
}
```

Создание и использование значений структуры

Рассмотрим следующие `struct` определения:

```
struct Foo {
    my_bool: bool,
    my_num: isize,
    my_string: String,
}
struct Bar (bool, isize, String);
struct Baz;
```

Построение новых структурных значений для этих типов является простым:

```
let foo = Foo { my_bool: true, my_num: 42, my_string: String::from("hello") };
let bar = Bar(true, 42, String::from("hello"));
let baz = Baz;
```

Поля доступа к структуре с использованием `.` :

```
assert_eq!(foo.my_bool, true);
assert_eq!(bar.0, true); // tuple structs act like tuples
```

Изменчивое связывание с структурой может иметь свои мутации:

```
let mut foo = foo;
foo.my_bool = false;
let mut bar = bar;
```

```
bar.0 = false;
```

Возможности сопоставления шаблонов Rust также могут использоваться для просмотра внутри `struct` :

```
// creates bindings mb, mn, ms with values of corresponding fields in foo
let Foo { my_bool: mb, my_num: mn, my_string: ms } = foo;
assert_eq!(mn, 42);
// .. allows you to skip fields you do not care about
let Foo { my_num: mn, .. } = foo;
assert_eq!(mn, 42);
// leave out `: variable` to bind a variable by its field name
let Foo { my_num, .. } = foo;
assert_eq!(my_num, 42);
```

Или создайте структуру, используя вторую структуру как «шаблон» с *синтаксисом обновления* Rust:

```
let foo2 = Foo { my_string: String::from("world"), .. foo };
assert_eq!(foo2.my_num, 42);
```

Методы структуры

Чтобы объявить методы в структуре (т. `impl` Функции, которые можно называть «на» `struct` или значения этого типа `struct`), создайте блок `impl` :

```
impl Foo {
    fn fiddle(&self) {
        // "self" refers to the value this method is being called on
        println!("fiddling {}", self.my_string);
    }
}

// ...
foo.fiddle(); // prints "fiddling hello"
```

`&self` здесь указывает на неизменную ссылку на экземпляр `struct Foo` чтобы вызвать метод `fiddle` . Если бы мы захотели изменить экземпляр (например, изменить одно из его полей), мы вместо этого возьмем `&mut self` (т. Е. Изменяемую ссылку):

```
impl Foo {
    fn tweak(&mut self, n: isize) {
        self.my_num = n;
    }
}

// ...
foo.tweak(43);
assert_eq!(foo.my_num, 43);
```

Наконец, мы могли бы также использовать `self` (обратите внимание на отсутствие `&`) в

качестве приемника. Это требует, чтобы экземпляр принадлежал вызывающему абоненту и заставил экземпляр перемещаться при вызове метода. Это может быть полезно, если мы хотим потреблять, уничтожать или иным образом полностью преобразовывать существующий экземпляр. Одним из примеров такого прецедента является обеспечение «цепных» методов:

```
impl Foo {
    fn double(mut self) -> Self {
        self.my_num *= 2;
        self
    }
}

// ...
foo.my_num = 1;
assert_eq!(foo.double().double().my_num, 4);
```

Обратите внимание, что мы также префикс `self` с `mut` чтобы мы могли мутировать себя, прежде чем возвращать его снова. Обратный тип `double` метода также требует некоторого объяснения. `Self` внутри блока `impl` ссылается на тип, к `impl` применяется `impl` (в данном случае `Foo`). Здесь в основном полезно сокращать, чтобы избежать повторной печати сигнатуры типа, но в свойствах он может использоваться для обозначения базового типа, реализующего определенный признак.

Чтобы объявить *связанный метод* (обычно называемый «метод класса» на других языках) для `struct` просто оставить аргумент `self` . Такие методы вызывают сам тип `struct` , а не его экземпляр:

```
impl Foo {
    fn new(b: bool, n: isize, s: String) -> Foo {
        Foo { my_bool: b, my_num: n, my_string: s }
    }
}

// ...
// :: is used to access associated members of the type
let x = Foo::new(false, 0, String::from("nil"));
assert_eq!(x.my_num, 0);
```

Обратите внимание, что методы структуры могут быть определены только для типов, объявленных в текущем модуле. Кроме того, как и в случае с полями, все методы структуры являются закрытыми по умолчанию и поэтому могут быть вызваны только кодом в том же модуле. Вы можете префиксные определения с ключевым словом `pub` чтобы сделать их вызываемыми из другого места.

Общие структуры

Структуры могут быть сделаны универсальными по одному или нескольким параметрам типа. Эти типы указаны в `<>` при обращении к типу:

```
struct Gen<T> {
    x: T,
    z: isize,
}

// ...
let _: Gen<bool> = Gen{x: true, z: 1};
let _: Gen<isize> = Gen{x: 42, z: 2};
let _: Gen<String> = Gen{x: String::from("hello"), z: 3};
```

Множественные типы могут быть заданы с помощью запятой:

```
struct Gen2<T, U> {
    x: T,
    y: U,
}

// ...
let _: Gen2<bool, isize> = Gen2{x: true, y: 42};
```

Параметры типа являются частью типа, поэтому две переменные одного и того же базового типа, но с разными параметрами, не являются взаимозаменяемыми:

```
let mut a: Gen<bool> = Gen{x: true, z: 1};
let b: Gen<isize> = Gen{x: 42, z: 2};
a = b; // this will not work, types are not the same
a.x = 42; // this will not work, the type of .x in a is bool
```

Если вы хотите написать функцию, которая принимает `struct` независимо от назначения ее типа, эта функция также должна быть сделана общей:

```
fn hello<T>(g: Gen<T>) {
    println!("{}", g.z); // valid, since g.z is always an isize
}
```

Но что, если мы хотим написать функцию, которая всегда может печатать `gx` ? Нам нужно было бы ограничить `T` типом, который может отображаться. Мы можем сделать это с ограничениями типа:

```
use std::fmt;
fn hello<T: fmt::Display>(g: Gen<T>) {
    println!("{} {}", g.x, g.z);
}
```

Функция `hello` теперь определена *только* для экземпляров `Gen` , тип `T` которых реализует `fmt::Display` . Если мы попытались передать в `Gen<(bool, isize)>` например, компилятор будет жаловаться на то, что `hello` не определено для этого типа.

Мы также можем использовать ограничения типов непосредственно по параметрам типа `struct` чтобы указать, что вы можете только построить эту `struct` для определенных типов:

```
use std::hash::Hash;
struct GenB<T: Hash> {
    x: T,
}
```

Любая функция, которая имеет доступ к `GenB` теперь знает, что тип `x` реализует `Hash` и, следовательно, может вызывать `.x.hash()` . Множественные ограничения типа для одного и того же параметра можно задать, разделив их на `+` .

То же, что и для функций, границы типов могут быть размещены после `<>` с использованием ключевого слова `where` :

```
struct GenB<T> where T: Hash {
    x: T,
}
```

Это имеет то же смысловое значение, но может упростить чтение и форматирование подписи при наличии сложных границ.

Параметры типа также доступны для методов экземпляра и связанных методов `struct` :

```
// note the <T> parameter for the impl as well
// this is necessary to say that all the following methods only
// exist within the context of those type parameter assignments
impl<T> Gen<T> {
    fn inner(self) -> T {
        self.x
    }
    fn new(x: T) -> Gen<T> {
        Gen{x: x}
    }
}
```

Если у вас есть ограничения типа на `Gen` 's `T` , они также должны отражаться в границах типа `impl` . Вы также можете сделать ограничения `impl` более жесткими, чтобы сказать, что данный метод существует только в том случае, если тип удовлетворяет определенному свойству:

```
impl<T: Hash + fmt::Display> Gen<T> {
    fn show(&self) {
        println!("{}", self.x);
    }
}

// ...
Gen{x: 42}.show(); // works fine
let a = Gen{x: (42, true)}; // ok, because (isize, bool): Hash
a.show(); // error: (isize, bool) does not implement fmt::Display
```

Прочитайте сооружения онлайн: https://riptutorial.com/ru/rust/topic/4583/сооружения

глава 43: Соответствие шаблону

Синтаксис

- _ // шаблон шаблона, соответствует чему-либо[1]
- *идентификатор* // шаблон привязки, сопоставляет все и связывает его с *идентификатором* [1]
- *ident* @ *pat* // то же, что и выше, но позволяет дополнительно сопоставлять то, что привязано
- ref *ident* // шаблон привязки, сопоставляет что-либо и связывает его с ссылочным *идентификатором* [1]
- ref mut *ident* // шаблон привязки, сопоставляет все и связывает его с изменяемым ссылочным *идентификатором* [1]
- & *pat* // соответствует ссылке (поэтому *pat* не является ссылкой, но рефери) [1]
- & mut *pat* // то же, что и выше, с изменяемой ссылкой[1]
- *CONST* // соответствует именованной константе
- *Struct {поле1, Field2}* // спички и разбирает значение структуры, см ниже замечание о fields[1]
- *EnumVariant* // соответствует варианту перечисления
- *EnumVariant* (*pat1* , *pat2*) // соответствует варианту перечисления и соответствующим параметрам
- *EnumVariant* (*pat1* , *pat2* , .., *patn*) // то же, что и выше, но пропускает все, кроме первого, второго и последнего параметров
- (*pat1* , *pat2*) // соответствует кортежу и соответствующим элементам[1]
- (*pat1* , *pat2* , .., *patn*) // то же, что и выше, но пропускает все, кроме первого, второго и последнего элементов[1]
- *lit* // соответствует литеральной константе (char, числовые типы, логические и строковые)
- *pat1* ... *pat2* // соответствует значению в этом (*включенном*) диапазоне (char и числовые типы)

замечания

При деконструкции структурного значения поле должно быть либо типа поля `field_name` либо `field_name : pattern` . Если шаблон не указан, выполняется неявное связывание:

```
let Point { x, y } = p;
// equivalent to
let Point { x: x, y: y } = p;

let Point { ref x, ref y } = p;
// equivalent to
let Point { x: ref x, y: ref y } = p;
```

1: необратимый шаблон

Examples

Согласование шаблонов с привязками

Можно привязывать значения к именам, используя `@` :

```
struct Badger {
    pub age: u8
}

fn main() {
    // Let's create a Badger instances
    let badger_john = Badger { age: 8 };

    // Now try to find out what John's favourite activity is, based on his age
    match badger_john.age {
        // we can bind value ranges to variables and use them in the matched branches
        baby_age @ 0...1 => println!("John is {} years old, he sleeps a lot", baby_age),
        young_age @ 2...4 => println!("John is {} years old, he plays all day", young_age),
        adult_age @ 5...10 => println!("John is {} years old, he eats honey most of the time",
adult_age),
        old_age => println!("John is {} years old, he mostly reads newspapers", old_age),
    }
}
```

Это напечатает:

```
John is 8 years old, he eats honey most of the time
```

Базовое совпадение

```
// Create a boolean value
let a = true;

// The following expression will try and find a pattern for our value starting with
// the topmost pattern.
// This is an exhaustive match expression because it checks for every possible value
match a {
  true => println!("a is true"),
  false => println!("a is false")
}
```

Если мы не рассмотрим все случаи, мы получим ошибку компилятора:

```
match a {
  true => println!("most important case")
}
// error: non-exhaustive patterns: `false` not covered [E0004]
```

Мы можем использовать _ в качестве аргумента default / wildcard, он соответствует всем:

```
// Create an 32-bit unsigned integer
let b: u32 = 13;

match b {
  0 => println!("b is 0"),
  1 => println!("b is 1"),
  _ => println!("b is something other than 0 or 1")
}
```

В этом примере будет напечатан:

```
a is true
b is something else than 0 or 1
```

Согласование нескольких шаблонов

Можно обрабатывать несколько разных значений одинаковым образом, используя `|` :

```
enum Colour {
    Red,
    Green,
    Blue,
    Cyan,
    Magenta,
    Yellow,
    Black
}

enum ColourModel {
    RGB,
    CMYK
}

// let's take an example colour
let colour = Colour::Red;

let model = match colour {
    // check if colour is any of the RGB colours
    Colour::Red | Colour::Green | Colour::Blue => ColourModel::RGB,
    // otherwise select CMYK
    _ => ColourModel::CMYK,
};
```

Условное сопоставление шаблонов с защитой

Шаблоны могут быть сопоставлены на основе значений, не зависящих от сопоставления значения с использованием, `if` защита:

```
// Let's imagine a simplistic web app with the following pages:
enum Page {
  Login,
  Logout,
```

```
  About,
  Admin
}

// We are authenticated
let is_authenticated = true;

// But we aren't admins
let is_admin = false;

let accessed_page = Page::Admin;

match accessed_page {
    // Login is available for not yet authenticated users
    Page::Login if !is_authenticated => println!("Please provide a username and a password"),

    // Logout is available for authenticated users
    Page::Logout if is_authenticated => println!("Good bye"),

    // About is a public page, anyone can access it
    Page::About => println!("About us"),

    // But the Admin page is restricted to administators
    Page::Admin if is_admin => println!("Welcome, dear administrator"),

    // For every other request, we display an error message
    _ => println!("Not available")
}
```

Появится сообщение *«Недоступно»* .

если пусть / пусть пусть

---

if let

Объединяет `match` шаблонов и оператор `if` и позволяет выполнять короткие неисчерпывающие совпадения.

```
if let Some(x) = option {
    do_something(x);
}
```

Это эквивалентно:

```
match option {
    Some(x) => do_something(x),
    _ => {},
}
```

Эти блоки также могут содержать инструкции `else` .

```
if let Some(x) = option {
```

```
    do_something(x);
} else {
    panic!("option was None");
}
```

Этот блок эквивалентен:

```
match option {
    Some(x) => do_something(x),
    None => panic!("option was None"),
}
```

---

while let

Объединяет совпадение шаблонов и цикл while.

```
let mut cs = "Hello, world!".chars();
while let Some(x) = cs.next() {
    print("{}+", x);
}
println!("");
```

Это печатает `H+e+l+l+o+,+ +w+o+r+l+d+!+` .

Это эквивалентно использованию `loop {}` и `match` заявление:

```
let mut cs = "Hello, world!".chars();
loop {
    match cs.next() {
        Some(x) => print("{}+", x),
        _ => break,
    }
}
println!("");
```

Извлечение ссылок из шаблонов

Иногда необходимо иметь возможность извлекать значения из объекта, используя только ссылки (т. Е. Без передачи права собственности).

```
struct Token {
  pub id: u32
}

struct User {
  pub token: Option<Token>
}


fn main() {
    // Create a user with an arbitrary token
    let user = User { token: Some(Token { id: 3 }) };
```

```
    // Let's borrow user by getting a reference to it
    let user_ref = &user;

    // This match expression would not compile saying "cannot move out of borrowed
    // content" because user_ref is a borrowed value but token expects an owned value.
    match user_ref {
        &User { token } => println!("User token exists? {}", token.is_some())
    }

    // By adding 'ref' to our pattern we instruct the compiler to give us a reference
    // instead of an owned value.
    match user_ref {
        &User { ref token } => println!("User token exists? {}", token.is_some())
    }

    // We can also combine ref with destructuring
    match user_ref {
        // 'ref' will allow us to access the token inside of the Option by reference
        &User { token: Some(ref user_token) } => println!("Token value: {}", user_token.id ),
        &User { token: None } => println!("There was no token assigned to the user" )
    }

    // References can be mutable too, let's create another user to demonstrate this
    let mut other_user = User { token: Some(Token { id: 4 }) };

    // Take a mutable reference to the user
    let other_user_ref_mut = &mut other_user;

    match other_user_ref_mut {
        // 'ref mut' gets us a mutable reference allowing us to change the contained value
directly.
        &mut User { token: Some(ref mut user_token) } => {
            user_token.id = 5;
            println!("New token value: {}", user_token.id )
        },
        &mut User { token: None } => println!("There was no token assigned to the user" )
    }
}
```

Он напечатает это:

```
User token exists? true
Token value: 3
New token value: 5
```

Прочитайте Соответствие шаблону онлайн: https://riptutorial.com/ru/rust/topic/1188/соответствие-шаблону

глава 44: Струны

Вступление

В отличие от многих других языков, Rust имеет **два** основных типа строк: `String` (тип строки, выделенной кучей) и `&str` (**заимствованная** строка, которая не использует дополнительную память). Знать разницу и когда использовать каждый, важно понять, как работает Руста.

Examples

Базовая манипуляция строками

```
fn main() {
    // Statically allocated string slice
    let hello = "Hello world";

    // This is equivalent to the previous one
    let hello_again: &'static str = "Hello world";

    // An empty String
    let mut string = String::new();

    // An empty String with a pre-allocated initial buffer
    let mut capacity = String::with_capacity(10);

    // Add a string slice to a String
    string.push_str("foo");

    // From a string slice to a String
    // Note: Prior to Rust 1.9.0 the to_owned method was faster
    // than to_string. Nowadays, they are equivalent.
    let bar = "foo".to_owned();
    let qux = "foo".to_string();

    // The String::from method is another way to convert a
    // string slice to an owned String.
    let baz = String::from("foo");

    // Coerce a String into &str with &
    let baz: &str = &bar;
}
```

Примечание. Оба метода `String::new` и `String::with_capacity` будут создавать пустые строки. Однако последний выделяет начальный буфер, что делает его изначально медленнее, но помогает сократить последующие распределения. Если конечный размер String известен, `String::with_capacity` следует использовать `String::with_capacity` .

Строчная нарезка

```
fn main() {
    let english = "Hello, World!";

    println!("{}", &english[0..5]); // Prints "Hello"
    println!("{}", &english[7..]);  // Prints "World!"
}
```

Обратите внимание, что нам нужно использовать оператор `&` . Он принимает ссылку и, таким образом, дает компилятору информацию о размере типа среза, который ему нужно распечатать. Без справки, два `println!` вызовы будут ошибкой времени компиляции.

Предупреждение: Нарезка работает **смещением байта** , а не смещением символа, и будет паниковать, если границы не находятся на границе символа:

```
fn main() {
    let icelandic = "Halló, heimur!"; // note that “ó” is two-byte long in UTF-8

    println!("{}", &icelandic[0..6]); // Prints "Halló", “ó” lies on two bytes 5 and 6
    println!("{}", &icelandic[8..]);  // Prints "heimur!", the “h” is the 8th byte, but the
7th char
    println!("{}", &icelandic[0..5]); // Panics!
}
```

Это также является причиной того, что строки не поддерживают простую индексацию (например, `icelandic[5]`).

Разделить строку

```
let strings = "bananas,apples,pear".split(",");
```

`split` возвращает итератор.

```
for s in strings {
  println!("{}", s)
}
```

И может быть «собран» в `Vec` с помощью `Iterator::collect` .

```
let strings: Vec<&str> = "bananas,apples,pear".split(",").collect(); // ["bananas", "apples",
"pear"]
```

От заемных до принадлежащих

```
// all variables `s` have the type `String`
let s = "hi".to_string();  // Generic way to convert into `String`. This works
                           // for all types that implement `Display`.

let s = "hi".to_owned();   // Clearly states the intend of obtaining an owned object

let s: String = "hi".into();       // Generic conversion, type annotation required
```

```
let s: String = From::from("hi");  // in both cases!

let s = String::from("hi");  // Calling the `from` impl explicitly -- the `From`
                             // trait has to be in scope!

let s = format!("hi");       // Using the formatting functionality (this has some
                             // overhead)
```

Помимо `format!()` Все вышеперечисленные методы одинаково быстры.

Разрыв длинномерных литералов

Разрыв регулярных строковых литералов с символом `\`

```
let a = "foobar";
let b = "foo\
         bar";

// `a` and `b` are equal.
assert_eq!(a,b);
```

Перерыв строк строки для разделения строк, и присоединиться к ним с `concat!` макрос

```
let c = r"foo\bar";
let d = concat!(r"foo\", r"bar");

// `c` and `d` are equal.
assert_eq!(c, d);
```

Прочитайте Струны онлайн: https://riptutorial.com/ru/rust/topic/998/струны

глава 45: Сырые указатели

Синтаксис

- let raw_ptr = & pointee как * const *type* // создать постоянный необработанный указатель на некоторые данные
- let raw_mut_ptr = & mut pointee как * mut *type* // создать изменяемый необработанный указатель на некоторые изменчивые данные
- let deref = * raw_ptr // разыменовать необработанный указатель (требуется небезопасный блок)

замечания

- Исходные указатели не гарантируют указания на действительный адрес памяти и, как следствие, неосторожное использование может привести к неожиданным (и, возможно, фатальным) ошибкам.
- Любая нормальная ссылка Rust (например, `&my_object` где тип `my_object` равна T) будет принуждать к `*const T` Аналогично, изменяемые ссылки принуждают к `*mut T`
- Необработанные указатели не переносят права собственности (в отличие от значений Box)

Examples

Создание и использование постоянных исходных указателей

```
// Let's take an arbitrary piece of data, a 4-byte integer in this case
let some_data: u32 = 14;

// Create a constant raw pointer pointing to the data above
let data_ptr: *const u32 = &some_data as *const u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    let deref_data: u32 = *data_ptr;
    println!("Dereferenced data: {}", deref_data);
}
```

Вышеприведенный код будет выводить: `Dereferenced data: 14`

Создание и использование изменяемых исходных указателей

```
// Let's take a mutable piece of data, a 4-byte integer in this case
let mut some_data: u32 = 14;
```

```
// Create a mutable raw pointer pointing to the data above
let data_ptr: *mut u32 = &mut some_data as *mut u32;

// Note: creating a raw pointer is totally safe but dereferencing a raw pointer requires an
// unsafe block
unsafe {
    *data_ptr = 20;
    println!("Dereferenced data: {}", some_data);
}
```

Вышеприведенный код выведет: `Dereferenced data: 20`

Инициализация исходного указателя на нулевой

В отличие от обычных ссылок Rust, raw-указатели могут принимать нулевые значения.

```
use std::ptr;

// Create a const NULL pointer
let null_ptr: *const u16 = ptr::null();

// Create a mutable NULL pointer
let mut_null_ptr: *mut u16 = ptr::null_mut();
```

Цепной разыменования

Как и в C, Rust raw указатели могут указывать на другие необработанные указатели (которые, в свою очередь, могут указывать на дополнительные raw-указатели).

```
// Take a regular string slice
let planet: &str = "Earth";

// Create a constant pointer pointing to our string slice
let planet_ptr: *const &str = &planet as *const &str;

// Create a constant pointer pointing to the pointer
let planet_ptr_ptr: *const *const &str = &planet_ptr as *const *const &str;

// This can go on...
let planet_ptr_ptr_ptr = &planet_ptr_ptr as *const *const *const &str;

unsafe {
    // Direct usage
    println!("The name of our planet is: {}", planet);
    // Single dereference
    println!("The name of our planet is: {}", *planet_ptr);
    // Double dereference
    println!("The name of our planet is: {}", **planet_ptr_ptr);
    // Triple dereference
    println!("The name of our planet is: {}", ***planet_ptr_ptr_ptr);
}
```

Это выведет: `The name of our planet is: Earth` четыре раза.

Отображение исходных указателей

У Rust есть форматирование по умолчанию для типов указателей, которые могут использоваться для отображения указателей.

```
use std::ptr;

// Create some data, a raw pointer pointing to it and a null pointer
let data: u32 = 42;
let raw_ptr = &data as *const u32;
let null_ptr = ptr::null() as *const u32;

// the {:p} mapping shows pointer values as hexadecimal memory addresses
println!("Data address: {:p}", &data);
println!("Raw pointer address: {:p}", raw_ptr);
println!("Null pointer address: {:p}", null_ptr);
```

Это выведет что-то вроде этого:

```
Data address: 0x7fff59f6bcc0
Raw pointer address: 0x7fff59f6bcc0
Null pointer address: 0x0
```

Прочитайте Сырые указатели онлайн: https://riptutorial.com/ru/rust/topic/7270/сырые-указатели

глава 46: тесты

Examples

Проверить функцию

```
fn to_test(output: bool) -> bool {
    output
}

#[cfg(test)] // The module is only compiled when testing.
mod test {
    use super::to_test;

    // This function is a test function. It will be executed and
    // the test will succeed if the function exits cleanly.
    #[test]
    fn test_to_test_ok() {
        assert_eq!(to_test(true), true);
    }

    // That test on the other hand will only succeed when the function
    // panics.
    #[test]
    #[should_panic]
    fn test_to_test_fail() {
        assert_eq!(to_test(true), false);
    }
}
```

(Игровая площадка)

Запуск с `cargo test` .

Интеграционные тесты

`lib.rs` :

```
pub fn to_test(output: bool) -> bool {
    output
}
```

Каждый файл в папке `tests/` скомпилирован как отдельный ящик. `tests/integration_test.rs`

```
extern crate test_lib;
use test_lib::to_test;

#[test]
fn test_to_test(){
    assert_eq!(to_test(true), true);
}
```

Контрольные тесты

С помощью тестовых тестов вы можете тестировать и измерять скорость кода, однако эталонные тесты по-прежнему нестабильны. Чтобы включить тесты в вашем грузовом проекте, вам нужна ночная ржавчина, поставьте тесты интеграции на `benches/` папку в корне вашего проекта Cargo и запустите `cargo bench` .

Примеры из llogiq.github.io

```
extern crate test;
extern crate rand;

use test::Bencher;
use rand::Rng;
use std::mem::replace;

#[bench]
fn empty(b: &mut Bencher) {
    b.iter(|| 1)
}

#[bench]
fn setup_random_hashmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::HashMap::new();

    b.iter(|| { map.insert(rng.gen::<u8>() as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::new();

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}

#[bench]
fn setup_random_vecmap_cap(b: &mut Bencher) {
    let mut val : u32 = 0;
    let mut rng = rand::IsaacRng::new_unseeded();
    let mut map = std::collections::VecMap::with_capacity(256);

    b.iter(|| { map.insert((rng.gen::<u8>()) as usize, val); val += 1; })
}
```

Прочитайте тесты онлайн: https://riptutorial.com/ru/rust/topic/961/тесты

глава 47: Файловый ввод-вывод

Examples

Прочитать файл в целом как строку

```
use std::fs::File;
use std::io::Read;

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode.
    match File::open(filename) {
        // The file is open (no error).
        Ok(mut file) => {
            let mut content = String::new();

            // Read all the file content into a variable (ignoring the result of the
operation).
            file.read_to_string(&mut content).unwrap();

            println!("{}", content);

            // The file is automatically closed when is goes out of scope.
        },
        // Error handling.
        Err(error) => {
            println!("Error opening file {}: {}", filename, error);
        },
    }
}
```

Прочитать файл по строкам

```
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let filename = "src/main.rs";
    // Open the file in read-only mode (ignoring errors).
    let file = File::open(filename).unwrap();
    let reader = BufReader::new(file);

    // Read the file line by line using the lines() iterator from std::io::BufRead.
    for (index, line) in reader.lines().enumerate() {
        let line = line.unwrap(); // Ignore errors.
        // Show the line and its number.
        println!("{}. {}", index + 1, line);
    }
}
```

Запись в файл

```
use std::env;
use std::fs::File;
use std::io::Write;

fn main() {
    // Create a temporary file.
    let temp_directory = env::temp_dir();
    let temp_file = temp_directory.join("file");

    // Open a file in write-only (ignoring errors).
    // This creates the file if it does not exist (and empty the file if it exists).
    let mut file = File::create(temp_file).unwrap();

    // Write a &str in the file (ignoring the result).
    writeln!(&mut file, "Hello World!").unwrap();

    // Write a byte string.
    file.write(b"Bytes\n").unwrap();
}
```

Прочитайте файл как Vec

```
use std::fs::File;
use std::io::Read;

fn read_a_file() -> std::io::Result<Vec<u8>> {
    let mut file = try!(File::open("example.data"));

    let mut data = Vec::new();
    try!(file.read_to_end(&mut data));

    return Ok(data);
}
```

`std::io::Result<T>` является псевдонимом для `Result<T, std::io::Error>` .

Макрос `try!()` Возвращается из функции при ошибке.

`read_to_end()` - это метод `std::io::Read` trait, который должен быть явно `use` d.

`read_to_end()` не возвращает данные, которые он читает. Вместо этого он помещает данные в указанный контейнер.

Прочитайте Файловый ввод-вывод онлайн: https://riptutorial.com/ru/rust/topic/1307/файловый-ввод-вывод

глава 48: Фьючерсы и Асинхронный ввод-вывод

Вступление

futures-rs - это библиотека, которая реализует фьючерсы и потоки нулевой стоимости в Rust.

Основными концепциями фьючерсного ящика являются Future and Stream .

Examples

Создание будущего с помощью функции onehot

Есть некоторые общие `Future` реализации признака в фьючерсной обрешетке. Один из них реализован в модуле `futures::sync::oneshot` и доступен через функцию `futures::oneshot` :

```
extern crate futures;

use std::thread;
use futures::Future;

fn expensive_computation() -> u32 {
    // ...
    200
}

fn main() {
    // The oneshot function returns a tuple of a Sender and a Receiver.
    let (tx, rx) = futures::oneshot();

    thread::spawn(move || {
        // The complete method resolves a values.
        tx.complete(expensive_computation());
    });

    // The map method applies a function to a value, when it is resolved.
    let rx = rx.map(|x| {
        println!("{}", x);
    });

    // The wait method blocks current thread until the value is resolved.
    rx.wait().unwrap();
}
```

Прочитайте Фьючерсы и Асинхронный ввод-вывод онлайн:
https://riptutorial.com/ru/rust/topic/8595/фьючерсы-и-асинхронный-ввод-вывод

глава 49: Червь - деструкторы в ржавчине

замечания

Использование Drop Trait не означает, что он будет запускаться каждый раз. Хотя он будет работать при выходе из области видимости или разматывании, это может быть не всегда так, например, когда вызывается `mem::forget` .

Это связано с тем, что паника при размотке приводит к прерыванию программы. Он также может быть скомпилирован с включенным `Abort on Panic` .

Для получения дополнительной информации ознакомьтесь с книгой: https://doc.rust-lang.org/book/drop.html

Examples

Простая реализация

```
use std::ops::Drop;

struct Foo(usize);

impl Drop for Foo {
    fn drop(&mut self) {
        println!("I had a {}", self.0);
    }
}
```

Падение для очистки

```
use std::ops::Drop;

#[derive(Debug)]
struct Bar(i32);

impl Bar {
    fn get<'a>(&'a mut self) -> Foo<'a> {
        let temp = self.0; // Since we will also capture `self` we..
                           // ..will have to copy the value out first
        Foo(self, temp) // Let's take the i32
    }
}

struct Foo<'a>(&'a mut Bar, i32); // We specify that we want a mutable borrow..
                                  // ..so we can put it back later on

impl<'a> Drop for Foo<'a> {
    fn drop(&mut self) {
        if self.1 < 10 { // This is just an example, you could also just put..
                         // ..it back as is
```

```
            (self.0).0 = self.1;
        }
    }
}

fn main() {
    let mut b = Bar(0);
    println!("{:?}", b);
    {
        let mut a : Foo = b.get(); // `a` now holds a reference to `b`..
        a.1 = 2;                   // .. and will hold it until end of scope
    }                              // .. here

    println!("{:?}", b);
    {
        let mut a : Foo = b.get();
        a.1 = 20;
    }
    println!("{:?}", b);
}
```

Drop позволяет создавать простые и отказоустойчивые проекты.

Отказоустойчивость для управления отладочной памятью

Управление памятью во время выполнения с помощью Rc может быть очень полезным, но также может быть сложно обернуть голову, особенно если ваш код очень сложный, и на один экземпляр ссылаются десятки или даже сотни других типов во многих областях.

Написание черты Drop, которая включает `println!("Dropping StructName: {:?}", self);` может быть очень ценным для отладки, поскольку он позволяет вам точно видеть, когда исчерпаны сильные ссылки на экземпляр структуры.

Прочитайте Червь - деструкторы в ржавчине онлайн: https://riptutorial.com/ru/rust/topic/7233/червь---деструкторы-в-ржавчине

глава 50: Черты

Вступление

Черты - это способ описания «контракта», который должна реализовать `struct` . Черты обычно определяют сигнатуры методов, но также могут обеспечивать реализации на основе других методов признака, обеспечивая при этом *границы признаков* .

Для тех, кто знаком с объектно-ориентированным программированием, черты можно рассматривать как интерфейсы с некоторыми незначительными отличиями.

Синтаксис

- trait Trait {fn method (...) -> ReturnType; ...}
- trait Trait: Bound {fn method (...) -> ReturnType; ...}
- impl Trait для типа {fn method (...) -> ReturnType {...} ...}
- impl <T> Trait for T где T: Bounds {fn method (...) -> ReturnType {...} ...}

замечания

- Черты обычно сравниваются с интерфейсами, но важно провести различие между ними. В языках OO, таких как Java, интерфейсы являются неотъемлемой частью классов, которые расширяют их. В Rust компилятор ничего не знает о чертах структуры, если эти черты не используются.

Examples

ОСНОВЫ

Создание признака

```
trait Speak {
    fn speak(&self) -> String;
}
```

Внедрение признака

```
struct Person;
struct Dog;

impl Speak for Person {
    fn speak(&self) -> String {
```

```
        String::from("Hello.")
    }
}

impl Speak for Dog {
    fn speak(&self) -> String {
        String::from("Woof.")
    }
}

fn main() {
    let person = Person {};
    let dog = Dog {};
    println!("The person says {}", person.speak());
    println!("The dog says {}", dog.speak());
}
```

Статическая и динамическая отправка

Можно создать функцию, которая принимает объекты, реализующие определенный признак.

Статическая отправка

```
fn generic_speak<T: Speak>(speaker: &T) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person);
    generic_speak(&dog);
}
```

Здесь используется статическая диспетчеризация, что означает, что компилятор Rust создаст специализированные версии `generic_speak` для типов `Dog` и `Person` . Это поколение специализированных версий полиморфной функции (или любого полиморфного объекта) во время компиляции называется **Мономорфизацией** .

Динамическая отправка

```
fn generic_speak(speaker: &Speak) {
    println!("{0}", speaker.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    generic_speak(&person as &Speak);
```

```
    generic_speak(&dog); // gets automatically coerced to &Speak
}
```

Здесь в скомпилированном двоичном `generic_speak` существует только одна версия `generic_speak` , а вызов `generic_speak speak()` выполняется с помощью поиска vtable во время выполнения. Таким образом, использование динамической диспетчеризации приводит к более быстрой компиляции и меньшему размеру скомпилированного двоичного файла, хотя во время выполнения он немного медленнее.

Объекты типа `&Speak` или `Box<Speak>` называются **объектами признаков** .

Связанные типы

- Использовать связанный тип, когда между типом, реализующим признак, и связанным с ним типом существует взаимно-однозначная взаимосвязь.
- Иногда он также известен как *тип вывода* , так как это элемент, присвоенный типу, когда мы применяем к нему черту.

Создание

```
trait GetItems {
    type First;
//  ^~~~ defines an associated type.
    type Last: ?Sized;
//           ^~~~~~~~ associated types may be constrained by traits as well
    fn first_item(&self) -> &Self::First;
//                           ^~~~~~~~~~~ use `Self::` to refer to the associated type
    fn last_item(&self) -> &Self::Last;
//                          ^~~~~~~~~~ associated types can be used as function output...
    fn set_first_item(&mut self, item: Self::First);
//                                     ^~~~~~~~~~~  ... input, and anywhere.
}
```

РЕАЛИЗАЦИЯ

```
impl<T, U: ?Sized> GetItems for (T, U) {
    type First = T;
    type Last = U;
//              ^~~ assign the associated types
    fn first_item(&self) -> &Self::First { &self.0 }
    fn last_item(&self) -> &Self::Last { &self.1 }
    fn set_first_item(&mut self, item: Self::First) { self.0 = item; }
}

impl<T> GetItems for [T; 3] {
    type First = T;
    type Last = T;
    fn first_item(&self) -> &T { &self[0] }
//                            ^ you could refer to the actual type instead of `Self::First`
```

```
    fn last_item(&self) -> &T { &self[2] }
    fn set_first_item(&mut self, item: T) { self[0] = item; }
}
```

Ссылка на связанные типы

Если мы уверены, что тип `T` реализует `GetItems` например, в generics, мы могли бы просто использовать `T::First` для получения ассоциированного типа.

```
fn get_first_and_last<T: GetItems>(obj: &T) -> (&T::First, &T::Last) {
//                                              ^~~~~~~~ refer to an associated type
    (obj.first_item(), obj.last_item())
}
```

В противном случае вам нужно явно указать компилятору, который реализует тип, реализующий

```
let array: [u32; 3] = [1, 2, 3];
let first: &<[u32; 3] as GetItems>::First = array.first_item();
//          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~ [u32; 3] may implement multiple traits which many
//                                        of them provide the `First` associated type.
//                                        thus the explicit "cast" is necessary here.
assert_eq!(*first, 1);
```

Ограничение со связанными типами

```
fn clone_first_and_last<T: GetItems>(obj: &T) -> (T::First, T::Last)
    where T::First: Clone, T::Last: Clone
//  ^~~~ use the `where` clause to constraint associated types by traits
{
    (obj.first_item().clone(), obj.last_item().clone())
}

fn get_first_u32<T: GetItems<First=u32>>(obj: &T) -> u32 {
//                          ^~~~~~~~~~~ constraint associated types by equality
    *obj.first_item()
}
```

Методы по умолчанию

```
trait Speak {
    fn speak(&self) -> String {
        String::from("Hi.")
    }
}
```

Метод будет вызван по умолчанию, за исключением случаев, если он был перезаписан в блоке `impl` .

```
struct Human;
struct Cat;

impl Speak for Human {}

impl Speak for Cat {
    fn speak(&self) -> String {
        String::from("Meow.")
    }
}

fn main() {
    let human = Human {};
    let cat = Cat {};
    println!("The human says {}", human.speak());
    println!("The cat says {}", cat.speak());
}
```

Выход :

Человек говорит, Привет.

Кошка говорит Мяу.

Помещение границы по признаку

При определении нового признака можно обеспечить, чтобы типы, желающие реализовать этот признак, проверяли ряд ограничений или ограничений.

Взяв пример из стандартной библиотеки, черта `DerefMut` требует, чтобы тип сначала реализовал свою черту `Deref` сестры:

```
pub trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Это, в свою очередь, позволяет `DerefMut` использовать связанный тип `Target` определенный `Deref` .

Хотя синтаксис может напоминать наследование:

- он включает все связанные элементы (константы, типы, функции, ...) связанного признака
- это позволяет полиморфизм от `&DerefMut` до `&Deref`

Это по-разному:

- можно использовать время жизни (например, `'static`) как связанное
- невозможно переопределить элементы связанных объектов (даже не функции)

Поэтому лучше всего думать об этом как о отдельной концепции.

Типы нескольких связанных объектов

Также возможно добавить несколько типов объектов в функцию Static Dispatch .

```
fn mammal_speak<T: Person + Dog>(mammal: &T) {
    println!("{0}", mammal.speak());
}

fn main() {
    let person = Person {};
    let dog = Dog {};

    mammal_speak(&person);
    mammal_speak(&dog);
}
```

Прочитайте Черты онлайн: https://riptutorial.com/ru/rust/topic/1313/черты

кредиты

S. No	Главы	Contributors
1	Начало работы с Rust	Andy Hayden, ar-ms, Aurora0001, Community, Cormac O'Brien, D. Ataro, David Grayson, Eric Platon, gavinb, IceyEC, John, Jon Gjengset, Kellen, kennytm, Kevin Montrose, Lukabot, mmstick, Neikos, Pavel Strakhov, Shepmaster, Timidger, Tot Zam, Tshepang, Wolf, xfix, Yohaï Berreby
2	Loops	Andy Hayden, apopiak, Arrem, Aurora0001, JDemler, John, kennytm, Mario Carneiro, Matt Smith, Matthieu M., mcarton, Sanpi, Shepmaster, Timidger, Winger Sendon, YOU
3	PhantomData	Neikos
4	Regex	Aurora0001, vaartis
5	rustup	torkleyy
6	Serde	Aurora0001, dtolnay, kennytm
7	Авто-разыменования	Aurora0001, John, kennytm, Kornel, Timidger, vaartis, Winger Sendon
8	Аргументы командной строки	Aurora0001
9	вариант	antoyo, Arrem, Aurora0001, fxlae, Kornel, letmutx, mcarton, Shepmaster
10	Владение	Aurora0001, Jon Gjengset
11	Время жизни	antoyo, Cormac O'Brien, Jean Pierre Dudey, letmutx, xetra11
12	Вставные значения	BookOwl
13	Встроенная сборка	4444, Aurora0001
14	Генерация случайных чисел	Phil J. Laszkowicz
15	Глобалы	Cormac O'Brien, Jean Pierre Dudey, John, kennytm, Leo Tindall , mcarton
16	Голая	John, mmstick, SplittyDev

	металлическая ржавчина	
17	грузовой	Arrem, Aurora0001, Bo Lu, Charlie Egan, Cormac O'Brien, David Grayson, Enigma, John, Lukas Kalbertodt
18	Дженерики	Kornel, xea
19	Документация	Aurora0001, Cormac O'Brien, Hauleth
20	Железная веб-платформа	4444, Aurora0001, Phil J. Laszkowicz
21	Закрытие и лямбда-выражения	xea
22	Интерфейс внешних функций (FFI)	Aurora0001, John, Konstantin V. Salikhov
23	итераторы	Aurora0001, Chris Emerson, Hauleth, John, Lukas Kalbertodt, Matt Smith, Shepmaster
24	Кортеж	adelarsq, Ameo, Aurora0001, John, Jon Gjengset, Matthieu M.
25	макрос	Aurora0001, kennytm, Matt Smith
26	Массивы, векторы и срезы	antoyo, Aurora0001, John, Matthieu M.
27	Модули	Aurora0001, Cormac O'Brien, Dumindu Madunuwan, John, KokaKiwi, Kornel, Lu.nemec, xetra11
28	Небезопасные рекомендации	Aurora0001, John
29	Обработка ошибок	Cormac O'Brien, John, kennytm, Winger Sendon, xea
30	Обработка сигналов	Aurora0001, Jean Pierre Dudey, mmstick
31	Образцы конверсии	Cormac O'Brien, Matthieu M.
32	Объектно-ориентированная ржавчина	adelarsq, Aurora0001, Leo Tindall, Marco Alka, Matthieu M., s3rvac, Sorona, Timidger
33	Операторы и	Aurora0001, John, Matthieu M.

	перегрузка	
34	Паники и разматывания	Aurora0001, Leo Tindall, Timidger
35	параллелизм	Aurora0001, John, Ruud, xea, zrneely
36	Пользовательский вывод: «Макросы 1.1»	Vi.
37	Приложения для графического интерфейса пользователя	eddy, vaartis
38	Примитивные типы данных	John
39	Руководство по стилю ржавчины	Aurora0001, Cldfire, James Gilles, tversteeg
40	Связанные константы	Ameo, Aurora0001, Hauleth
41	Сеть TCP	E_net4
42	сооружения	4444, Jon Gjengset, letmutx
43	Соответствие шаблону	adelarsq, Andy Hayden, aSpex, Aurora0001, Cormac O'Brien, Lukas Kalbertodt, mcarton, mnoronha, xea
44	Струны	Arrem, Aurora0001, David Grayson, KokaKiwi, Lukas Kalbertodt, mcarton, rap-2-h, tmr232, Yos Riady
45	Сырые указатели	xea
46	тесты	Aurora0001, Cormac O'Brien, IceyEC, JDemler, Jean Pierre Dudey, kennytm, Lu.nemec, mcarton
47	Файловый ввод-вывод	antoyo, Kornel
48	Фьючерсы и Асинхронный ввод-вывод	KolesnichenkoDS
49	Червь -	Leo Tindall, Neikos

	деструкторы в ржавчине	
50	Черты	a10y, adelarsq, Arrem, Aurora0001, Cormac O'Brien, Hauleth, John, kennytm, Leo Tindall, Matt Smith, Matthieu M., Mylainos, RamenChef, SplittyDev, tversteeg, xea