

Практика использования и наиболее полезные конструкции языка Rust

Содержание

| | |
|---|----|
| 1 Ресурсы по языку Rust | 1 |
| 2 Установка Rust | 1 |
| 3 Вводные замечания | 2 |
| 4 Начало работы | 3 |
| 4.1 Первая программа на Rust | 4 |
| 5 Основы языка | 7 |
| 5.1 Числа | 7 |
| 5.2 Управление ходом выполнения программы | 8 |
| Список литературы | 10 |
| Список листингов | 10 |

1. Ресурсы по языку Rust

<https://www.rust-lang.org/tools>
<https://doc.rust-lang.org/book/>
<https://doc.rust-lang.org/stable/rust-by-example/>

2. Установка Rust

Установить Rust проще всего с помощью утилиты `rustup` – это установщик языка и менеджер версий. Для операционной системы Windows можно скачать `rustup-init.exe` со страницы проекта <https://www.rust-lang.org/learn/get-started>

Установить Rust на Linux можно так

```
$ curl https://sh.rustup.rs -sSf | bash
...
Current installation options:

default host triple: x86_64-unknown-linux-gnu
default toolchain: stable (default)
profile: default
modify PATH variable: yes

1) Proceed with installation (default)
```

```
2) Customize installation
3) Cancel installation
>1

info: profile set to 'default'
info: default host triple is x86_64-unknown-linux-gnu
info: syncing channel updates for 'stable-x86_64-unknown-linux-gnu'
...
```

Rust часто обновляется и чтобы получить последнюю версию, можно воспользоваться командой `rustup update`.

Собрать проект и обновить его зависимости можно с помощью утилиты `cargo`

```
cargo build # build your project
cargo run  # cargo run
cargo test # test project
cargo doc  # build documentation for your project
cargo publish # publish a library to crates.io
```

То есть `cargo` знает, как превратить Rust-код в исполняемый бинарный файл, а также может управлять процессом загрузки и компиляции проектных зависимостей.

3. Вводные замечания

Система владения устанавливает *время жизни* каждого значения, что делает ненужным сборку мусора в ядре языка и обеспечивает надежные, но вместе с тем гибкие интерфейсы для управления такими ресурсами, как сокеты и описатели файлов. Передача (*move*) позволяет передавать значение от одного владельца другому, а заимствование (*borrowing*) – использовать значение временно, не изменяя владельца.

Rust – типобезопасный язык. Но что понимается под типобезопасностью? Ниже приведено определение «неопределенного поведения» из стандарта языка C 1999 года, известного под названием «C99»: *неопределенное поведение – это поведение, являющееся следствием использования непереносимой или некорректной программной конструкции либо некорректных данных, для которого в настоящем Международном стандарте нет никаких требований.*

Рассмотрим следующую программу на C

```
int main(int argc, char **argv) {
    // объявление одноэлементного массива беззнаковых длинных целых чисел
    unsigned long a[1];
    // обращение к 4-ому элементу массива; индекс, нарушает границу диапазона
    a[3] = 0x7ffff7b36cebUL;
    return 0;
}
```

Эта программа обращается к элементу за концом массива `a`, поэтому согласно C99 ее *поведение не определено*, т.е. она может делать все что угодно. «Неопределенная» операция не просто возвращает неопределенный результат, она дает программе карт-бланш на произвольное выполнение(!).

C99 предоставляет компилятору такое право, чтобы он мог генерировать более быстрый код. Чем возлагать на компилятор ответственность за обнаружение и обработку странного поведения вроде выхода за конец массива, стандарт предполагает, что программист должен позаботиться о том, чтобы такие ситуации никогда не возникали.

Если программа написана так, что ни на каком пути выполнения *неопределенное выполнение невозможно*, то будем говорить, что программа *корректна* (well defined).

Если встроенные в язык проверки *гарантируют корректность программы*, то будем называть язык *типобезопасным* (type safe).

Тщательно написанная программа на C или C++ может оказаться типобезопасной, **но ни C, ни C++ не является типобезопасным языком**: в приведенном выше примере нет ошибок типизации, и тем не менее она демонстрирует неопределенное поведение. С другой стороны, **Python – типобезопасный язык**, его интерпретатор тратит время на обнаружение выхода за границы массива и обрабатывает его лучше, чем компилятор C.

4. Начало работы

Создать проект на Rust можно командой `cargo new <project_name>`

```
$ cargo new hello # создать проект hello
$ tree
.
hello/
  Cargo.toml
  src/
    main.rs
$ cd hello
$ cargo run # запустить проект
   Compiling hello v0.1.0 (/home/kosyachenko/Projects/GARBAGE/rust_projects/hello)
Finished dev [unoptimized + debuginfo] target(s) in 0.42s
Running `target/debug/hello`
Hello, world!
# Дерево проекта изменилось
$ tree
.
Cargo.lock # артефакт
Cargo.toml
src/
  main.rs
target/ # артефакт
  CACHEDIR.TAG
  debug/
    build
    deps/
      hello-27...
      hello-27...d
    examples/
      hello
      hello.d
    incremental/
      hello-imy.../
        s-ghim...
```

В основном каталоге имеется файл `Cargo.toml`, содержащий описание метаданных проекта, таких как имя проекта, его версия и его зависимости. Исходный код попадает в директорию `src`.

Выполнение команды `cargo run` привело также к добавлению к проекту новых файлов. Теперь у нас в основном каталоге проекта есть файл `Cargo.lock` и каталог `target`. В `Cargo.lock` указываются конкретные номера версий всех зависимостей, чтобы будущие сборки составлялись точно также, как и эта, пока содержимое `Cargo.toml` не изменится.

4.1. Первая программа на Rust

Нужно как обычно с помощью `cargo new hello` создать новый проект. Перейти в созданную директорию проекта и в файле `main.rs` директории `src` написать следующее

./src/main.rs

```
fn greet_world() {
    println!("Hello, world!");
    let southern_germany = "Germany";
    let japan = "Japan";
    let regions = [southern_germany, japan];

    for region in regions.iter() {
        println!("{}", &region);
    }
}

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

Восклицательный знак свидетельствует об использовании *макроса*. Для операции присваивания в Rust, которую правильнее было бы называть *привязкой переменной*, используется ключевое слово `let`. Поддержка Unicode предоставляется самим языком.

Для *литералов массива* используются *квадратные скобки*. Для возврата итератора метод `iter()` может присутствовать во многих типах. Амперсанд «заимствует» `region` так, чтобы доступ предоставлялся *только для чтения*.

Строки ганатировано получают кодировку UTF-8.

Пример

src/main.rs

```
fn main() { // (1)
    let penguin_data = "\
    // (2)
    common name,length (cm)
    Little penguin,33
    Yellow-eyed penguin,65
    Fiordland penguin,60
    Invalid,data
    ";

    let records = penguin_data.lines();

    for (i, record) in records.enumerate() {
        if i == 0 || record.trim().len() == 0 { // (3)
            continue;
        }

        let fields: Vec<_> = record // (4)
            .split(',') // (5)
            .map(|field| field.trim()) // (6)
            .collect(); // (7)
        if cfg!(debug_assertations) { // (8)
            eprintln!("debug: {:?} -> {:?}", record, fields); // (9)
        }

        let name = fields[0];
    }
}
```

```

        if let Ok(length) = fields[1].parse::<f32>() { // (10)
            println!("{}", {length} cm", name, length); // (11)
        }
    }
}

```

(1) – исполняемым проектам требуется функция `main()`. (2) – отключение завершающего символа новой строки. (3) – пропуск строки заголовка и строк, состоящих из одних пробелов. (4) – начало со строки текста. (5) – разбиение записи на поля. (6) – обрезка пробелов в каждом поле. (7) – Сборка набора полей. (8) – `cargo!` проверяет конфигурацию в процессе компиляции. (9) – `eprintln!` выводит данные на стандартное устройство сообщений об ошибках (`stderr`). (10) – попытка выполнения парсинга поля в виде числа с плавающей точкой. (11) – `println!` помещает данные на стандартное устройство вывода (`stdout`).

Переменная `fields` помечена типом `Vec<_>`. `Vec` – сокращение от `_vector_`, типа коллекции, способного динамически расширяться. Знак подчеркивания предписывает Rust вывести тип элемента.

На Python решение выглядело бы так

```

#!/python
import typing as t

def main():
    penguin_data: str = """
        common name, length (cm)
        Little penguin, 33
        Yellow-eyed penguin, 65
        Fiordland penguin, 60
        Invalid, data
    """

    records: t.List[str] = penguin_data.split("\n")

    for (i, record) in enumerate(records):
        if i == 0 or len(record.strip()) == 0:
            continue

        fields: t.List[str] = list(map(lambda field: field.strip(), records.split(",")
        # Или с помощью спискового включения
        # fields: t.List[str] = [record.strip() for record in records.split(",")]

        if __debug__:
            print(f"debug: {record} -> {fields}")

        name: str = fields[0]
        # На Rust это блок выглядит изящнее
        try:
            length = float(fields[1])
        except ValueError as err:
            continue
        else:
            print(f"{name}, {length} cm")

if __name__ == "__main__":
    main()

```

Макросы похожи на функции, но вместо возвращения данных они возвращают код. Макросы часто используются для упрощения общеупотребительных шаблонов. Поле заполнения `{}` заставляет Rust воспользоваться методом представления значения в виде строки, который определил программист, а не представлением по умолчанию, доступным при указании поля заполнителя `{:?}`.

`if let Ok(length) = fields[1].parse::<f32>()` читается так «попытаться разобрать `fields[1]` в виде 32-разрядного числа с плавающей точкой, и в случае успеха присвоить число переменной `length`».

Конструкция `if let` – краткий метод обработки данных, предоставляющий также локальную переменную, которой присваиваются эти данные. Метод `parse()` возвращает `Ok(T)` (где `T` означает любой тип), если ему удастся провести разбор строки; в противном случае он возвращает `Err(E)` (где `E` означает тип ошибки). Применение `if let Ok(T)` позволяет пропустить любые случаи ошибок, подобные той, что встречаются при обработке строки `Invalid,data`.

Когда Rust не способен вывести тип из окружающего контекста, он запрашивает конкретное указание. В вызов `parse()` включается встроенная аннотация типа в виде `parse::<f32>()`.

Преобразование исходного кода в исполняемый файл называется *компиляцией*.

В Rust-программах отсутствуют:

1. Висячие указатели – прямые ссылки на данные, ставшие недействительными в ходе выполнения программы,
2. Состояние гонки – неспособность из-за изменения внешних факторов определить, как программа будет вести себя от запуска к запуску,
3. Переполнение буфера – попытка обращения к 12-му элементу массива, состоящего из 6 элементов

В Rust *пустой `min: ()`* (произносится как «юнит»). Когда нет никакого другого значимого возвращаемого значения, выражение возвращает `()`.

Rust предлагает программистам детальный контроль над размещением структур данных в памяти и над схемами доступа к ним. Временами возникает острая потребность в управлении производительностью приложения. При этом важную роль может сыграть хранение данных в *стеке*, а не в *куче*.

Особые возможности Rust:

- Достижение высокой производительности,
- Выполнение одновременных (параллельных) вычислений,
- Достижение эффективной работы с памятью.

Rust позволяет воспользоваться всей доступной производительностью компьютера. Он не использует для обеспечения безопасности памяти сборщик мусора.

В Rust нет никакой глобальной блокировки интерпретатора, ограничивающей скорость потока.

Единицей компиляции программы на Rust является не отдельный файл, а целый пакет (известный как *крейт*). Поскольку крейты могут включать в себя несколько модулей, они могут становиться весьма большими объектами компиляции. Это конечно, позволяет оптимизировать весь крейт, но требует также его компиляции.

`let` используется для *привязки переменной*. По умолчанию переменные *неизменяемы*, то есть предназначены только для чтения, а не для чтения-записи.

5. Основы языка

5.1. Числа

Преобразования между типами всегда носят явный характер. В Rust у чисел могут быть методы: например, для округления 24.5 к ближайшему целому числу используется `24.5_f32.round()`, а не `round(24.5)`.

Литералы чисел с плавающей точкой без явно указанной аннотации типа становятся 32- или 64-разрядными в зависимости от контекста.

Имеющиеся в Rust требования к безопасности типов *не позволяют проводить сравнение между типами*. Например, следующий код не пройдет компиляцию:

```
fn main() {
    let a: i32 = 10;
    let b: u16 = 100;

    if a < b { // error[E0308]: mismatched types
        println!("Ten is less than one hundred.")
    }
}
```

Безопаснее всего привести меньший тип к большему (например, 16-разрядный тип к 32-разрядному): `(b as i32)`. Иногда это называют расширением.

Порой использовать ключевое слово `as` накладывает слишком большие ограничения. В следующем листинге показан Rust-метод, заменяющий ключевое слово `as` в тех случаях, когда приведение может дать сбой

```
use std::convert::TryInto;

fn main() {
    let a: i32 = 10;
    let b: u16 = 100;

    let b_ = b.try_into().unwrap(); // try_into() -> mun Result

    if a < b_ {
        println!("Ten is less than one hundred.");
    }
}
```

Ключевое слово `use` переносит типаж `std::convert::TryInto` в локальную область видимости. В результате этого происходит разблокирование метода `try_into()`, вызываемого в отношении переменной `b`.

Типаж можно рассматривать как абстрактные классы или интерфейсы. Метод `b.try_into()` возвращает значение типа `i32`, завернутое в значение типа `Result`. Значение успеха может быть обработано методом `unwrap()`, в результате чего здесь будет возвращено значение `b`, имеющее тип `i32`.

Rust включает ряд допуско, позволяющих сравнивать числовые значения с плавающей точкой. Эти допуски определяются как `f32::EPSILON` и `f64::EPSILON`.

Операции, выдающие математически неопределенные результаты, например извлечение квадратного корня из отрицательного числа, создают особые проблемы. Для обработки таких случаев в тип числа с плавающей точкой включены значения NaN – «not a number».

Чтобы добавить контейнер (крейт) в проект достаточно добавить в раздел `[dependencies]` имя крейта и его версию в файл `Cargo.toml`

Cargo.toml

```
...
[dependencies]
num = "0.4"
...
```

```
use num::complex::Complex;

fn main() {
    let a = Complex {re: 2.1, im: -1.2}; // у каждого типа в Rust имеется литеральный синтаксис
    let b = Complex::new(11.1, 22.2);
    let result = a + b;

    println!("{}", result.re, result.im); // доступ к полям через оператор точка
}
```

Ключевое слово `use` помещает тип `Complex` в локальную область видимости. В Rust нет конструкторов, вместо этого у каждого типа есть литеральная форма.

Инициализировать типы можно путем использования имени типа и присвоения его полям значений в фигурных скобках: `Complex { re: 2.1, im: -1.2 }`. Для упрощения программ метод `new()` реализован у многих типов. Но это соглашение не часть языка Rust.

Поддерживаются две форму инициализации неэлементарных типов:

- Литеральный синтаксис: `Complex { re: 2.1, im: -1.2 }`,
- Статическим методом `new()`: `Complex::new(11.1, 22.2)`.

Статический метод – это функция, доступная для *типа*, но не для *экземпляра типа*. В реальном коде предпочтительнее вторая форма.

5.2. Управление ходом выполнения программы

Базовая форма цикла `for` имеет следующий вид

```
for item in container {
    // ...
}
```

Эта базовая форма делает каждый последующий элемент в контейнере `container` доступным в качестве элемента `item`.

Несмотря, на то, что переменная `container` остается в локальной области видимости, теперь ее *время жизни* истекло. Rust считает, что раз блок закончился, то надобности в переменной `container` миновала.

Когда чуть позже в программе возникнет желание воспользоваться переменной `container` еще раз, следует воспользоваться указателем. Когда указатель опущен, Rust полагает, что переменная `container` больше не нужна. Чтобы добавить *указатель* на контейнер, нужно, как показано в следующем примере, поставить перед его именем знак амперсанда (`&`)

```
for item in &container {
    // ...
}
```


Если в ходе циклического перебора элементов нужно внести изменения в каждый элемент, можно воспользоваться *указателем, допускающим изменения*, включив в код ключевое слово `mut`

```
for item in &mut container {  
    // ...  
}
```

Безымянные циклы. Если в блоке не используется локальная переменная, то по соглашению применяется знак подчеркивания «_». Использование этой схемы в сочетании с синтаксисом *исключающего диапазона* (`n..m`) и синтаксисом *включающего диапазона* (`n..=m`) показывает, что целью является выполнение цикла фиксированное количество раз. Например

```
for _ in 0..10 {  
    // ...  
}
```

Ключевое слово `continue` действует вполне ожидаемым образом

```
// Вывести только нечетные  
for item in 0..10 {  
    if item % 2 == 0 {  
        continue;  
    }  
}
```

Прерывание цикла выполняется с помощью ключевого слова `break`. При этом Rust работает привычным образом

```
fn main() {  
    for (x, y) in (0..).zip(0..) { // zip работает на бесконечной последовательности  
        if x + y > 100 {  
            break;  
        }  
        println!("x={x}, y={y}", x, y);  
    }  
}
```

В Python пришлось бы организовывать бесконечный цикл `while`, например так

```
def main():  
    x, y = (0, 0)  
    while True:  
        if x + y > 100:  
            break  
        print(f"x={x}, y={y}")  
        x += 1  
        y += 1
```

Прерывание во вложенных циклах. Прервать выполнение вложенного цикла можно с помощью *меток циклов*. Метка цикла представляет собой идентификатор с префиксом в виде *апострофа* ,

```
'outer: for x in 0.. {  
    for y in 0.. {  
        for z in 0.. {  
            if x + y + z > 10 {  
                break 'outer;  
            }  
        }  
    }  
}
```

```
    // ...  
  }  
}  
}
```

Условное ветвление. `if` допускает применение любого выражения, вычисленного в булево значение (`true` или `false`). Когда нужно протестировать несколько значений, можно добавить цепочку блоков `if else`. Блок `else` соответствует всему, чему еще не нашлось соответствие. Например

```
if item == 42 {  
    // ...  
} else if item == 132 {  
    // ...  
} else {  
    // ...  
}
```

В Rust отсутствует концепция «правдивых» или «ложных» типов. В других языках (например, в Python) допускается, чтобы особые значения, например 0 или пустая строка, означали `false`, а другие значения означали `true`, но в Rust это не практикуется. Единственным значением, которое может быть `true`, является `true`, а за `false` может принимать только `false`.

Список литературы

1. *Кольцов Д.М.* Си на примерах. Практика, практика и только практика. – СПб.: Наука и Техника, 2019. – 288 с.

Листинги