### Функциональное программирование в Rust

#### 1. Неизменяемость и Передача по значению

В функциональном программировании данные рассматриваются как неизменяемые. В приведенном примере, переменная x неизменяема после передачи её в функцию double, что приводит к ошибке компиляции при попытке использования её после вызова функции.

fn main() {

let x = 5;

let y = double(x);

println!("x: {}, y: {}", x, y); // Ошибка компиляции: значение x больше недоступно

}

fn double(mut num: i32) -> i32 {

num \*= 2;

num

}

#### 2. Функции Высших Порядков

Пример демонстрирует использование функции высшего порядка map, которая принимает вектор и функцию-маппер. Это позволяет применять функцию к каждому элементу вектора.

fn main() {

let numbers = vec![1, 2, 3, 4, 5];

let squared: Vec<i32> = map(numbers, |x| x \* x);

println!("{:?}", squared); // Вывод: [1, 4, 9, 16, 25]

}

fn map<T, F>(input: Vec<T>, mapper: F) -> Vec<T>

where

F: Fn(T) -> T,

{

input.into\_iter().map(mapper).collect()

}

#### 3. Замыкания (Closures)

Замыкания в Rust представляют собой анонимные функции, которые могут захватывать и использовать переменные из окружающего контекста. Они позволяют писать компактный и выразительный код.

Пример:

fn main() {

let add = |a, b| a + b;

let result = add(3, 5);

println!("Результат сложения: {}", result); // Вывод: Результат сложения: 8

}

#### 4. Рекурсия и Хвостовая Рекурсия

Функциональное программирование часто использует рекурсию. Rust поддерживает хвостовую рекурсию, что позволяет оптимизировать рекурсивные вызовы и избежать переполнения стека.

Пример:

fn factorial(n: u64) -> u64 {

fn factorial\_tail\_recursive(n: u64, acc: u64) -> u64 {

if n == 0 {

acc

} else {

factorial\_tail\_recursive(n - 1, acc \* n)

}

}

factorial\_tail\_recursive(n, 1)

}

fn main() {

let result = factorial(5);

println!("Факториал числа 5: {}", result); // Вывод: Факториал числа 5: 120

}

#### 5. Сопоставление с образцом и деконструкция

Сопоставление с образцом в Rust позволяет компактно и выразительно обрабатывать структуры данных, перечисления и другие типы. Деконструкция значений делает код более читаемым и понятным.

Пример:

struct Point {

x: i32,

y: i32,

}

fn main() {

let point = Point { x: 10, y: 20 };

match point {

Point { x, y } => println!("Координаты: x = {}, y = {}", x, y),

}

}

#### 6. Функции и Функциональные Комбинаторы

Функциональные комбинаторы представляют собой высокоуровневые функции, которые комбинируют другие функции. В Rust, использование функциональных комбинаторов может улучшить читаемость кода.

Пример:

fn main() {

let numbers = vec![1, 2, 3, 4, 5];

let sum: i32 = numbers.iter().fold(0, |acc, &x| acc + x);

let doubled: Vec<i32> = numbers.iter().map(|x| x \* 2).collect();

println!("Сумма: {}", sum); // Вывод: Сумма: 15

println!("Удвоенные значения: {:?}", doubled); // Вывод: Удвоенные значения: [2, 4, 6, 8, 10]

}

### Шаблонное программирование в Rust

#### 1. Обобщенные Типы (Generics)

Использование обобщенных типов позволяет создавать структуры данных, которые могут работать с различными типами данных, без необходимости дублирования кода.

struct Point<T> {

x: T,

y: T,

}

fn main() {

let int\_point = Point { x: 1, y: 2 };

let float\_point = Point { x: 1.5, y: 2.5 };

println!("{:?}", int\_point);

println!("{:?}", float\_point);

}

#### 2. Трейты (Traits)

Трейты в Rust предоставляют интерфейсы для типов, позволяя им реализовывать общие методы. В этом примере Printable трейт определяет метод print, который затем реализуется для структуры Message.

trait Printable {

fn print(&self);

}

struct Message {

content: String,

}

impl Printable for Message {

fn print(&self) {

println!("{}", self.content);

}

}

fn main() {

let message = Message {

content: "Hello, Rust!".to\_string(),

};

message.print();

}

#### 3. Производные Типы (Associated Types)

В Rust, производные типы позволяют структурам данных и типам иметь ассоциированные типы, что обеспечивает более гибкую параметризацию.

Пример:

trait Container {

type Item;

fn get\_item(&self) -> Self::Item;

}

struct Wrapper(i32);

impl Container for Wrapper {

type Item = i32;

fn get\_item(&self) -> Self::Item {

self.0

}

}

fn main() {

let wrapper = Wrapper(42);

println!("Item: {}", wrapper.get\_item()); // Вывод: Item: 42

}

#### 4. Оператор where для Ограничения Типов

Оператор where используется для улучшения читаемости и управления ограничениями типов в обобщенных функциях и структурах.

Пример:

struct Container<T> {

item: T,

}

impl<T> Container<T> where T: std::fmt::Debug {

fn print\_item(&self) {

println!("{:?}", self.item);

}

}

fn main() {

let container = Container { item: 42 };

container.print\_item(); // Вывод: 42

}

### Заключение

Rust предоставляет разработчикам инструменты функционального и шаблонного программирования, поддерживая неизменяемость данных, передачу по значению, обобщенные типы и трейты. Эти концепции способствуют созданию безопасного и выразительного кода, делая Rust привлекательным для различных областей программирования.