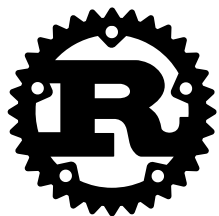






Rust 짱어먹기

Let's get Rusty



Ferris	Meaning
	This code does not compile!
	This code panics!
	This code block contains unsafe code.
	This code does not produce the desired behavior.

목차

1. Rust의 강점
2. Rust의 단점 (싫어하는 이유)
3. 활용 예시 코드 (Result와 Match)

만들어진 계기



Graydon Hoare



Rust의 강점

1. 안전한 메모리 관리 (Ownership)
 - ex) 변수가 스코프 밖으로 벗어나면 값이 버려짐(drop 호출)
2. 불변성
 - `let` or `let mut`
3. 철저한 예외나 에러 관리 (Result, match)
 - `Ok()`, `Err()`, ***panic!*** 명시적으로 작성
4. 눈물 없이는 볼 수 없는 **감동적인 컴파일 에러 메시지**
 - 아래 예제에서 느낄 수 있습니다.

```
error[E0308]: mismatched types
--> src\client\tenor_client.rs:39:17
39 | |                                cod: 500,
    | |                                ^^^ help: a field with a similar name exists: `code`
```

메모리와 할당

```
{  
    let s = String::from("hello"); // s는 여기서부터 유효합니다  
  
    // s를 가지고 뭔가 합니다  
}  
// 이 스코프는 끝났고, s는 더 이상  
// 유효하지 않습니다
```

1. 런타임에 운영체제로부터 메모리가 요청되어야 한다.
2. String의 사용이 끝났을 때 운영체제에게 메모리를 반납할 방법이 필요하다.

`String::from` 은 힙에 생성됩니다.

러스트는 `}` 괄호가 닫힐때 자동적으로 `drop` 을 호출합니다.

변수 선언, 불변성

1. 기본 변수는 immutable (불변)
 - 가변은 *가끔* 값을 나중에 변경하면 찾기가 어려움
2. let 키워드를 사용하여 타입 자동 추론
 - 변수 타입에 대한 오류를 미리 감지함

변수의 불변성

- 기본 변수는 immutable (불변)

```
fn main() {  
    let x = 5; // let mut x = 5;  
    println!("The value of x is: {}", x);  
    x = 6;  
    println!("The value of x is: {}", x);  
}
```

error[E0384]: re-assignment of immutable variable `x`

--> src/main.rs:4:5

```
2 |         let x = 5;  
  |         - first assignment to `x`  
3 |         println!("The value of x is: {}", x);  
4 |         x = 6;  
  |         ^^^^^ re-assignment of immutable variable
```

자동 추론 (타입 추론)

- let 키워드를 사용하여 타입 자동 추론

```
let unsigned_int = 123_u32; // type u32
```

```
let a: u64 = 123; // type u64
```

```
let pi = 3.14159265358979323846264338327950288; // type f64
```

```
let small_pi : f32 = 3.14; // type f32
```

```
let url = "https://httpbin.org/ip"; // type &str
```

```
let tenor_key = env::var("TENOR_API_KEY")  
    .unwrap_or_else(|_| String::from("<default_api_key>")); // type String
```

```
let signed_int = 0xff_ff_ff_ff_ff; // type i32 ???
```


Ownership 잠깐 보기

- 힙에 생성되는 변수를 다른 변수에 할당할 경우
- Ownership은 복사되지 않고 (Not Copy)
- 이동(Move) 된다.
- 즉, Rust는 각각의 값은 해당 값의 Owner라고 불리는 변수를 딱 하나 가지고 있다.
- 할당할 때마다 함수에 넘겨주거나, 반환받는건 너무 불편하다!
 - *Reference/Borrow*

Ownership 예제 코드

```
pub fn fail_move_ownership() {  
    let i_am_on_stack: &str = "7427466391.com";  
    let me_too = i_am_on_stack;  
  
    println!("i_am_on_stack is {}", i_am_on_stack);  
    println!("me_too is {}", me_too);  
  
    let i_am_on_heap = String::from("Ferris");  
    print_function(i_am_on_heap);  
  
    let me_too = i_am_on_heap;  
    print_function(me_too);  
}  
  
fn print_function(name: String) {  
    println!("{}", name);  
}
```

error[E0382]: use of moved value: `i_am_on_heap`
--> src\ownership\ownership.rs:11:18

```
8 |     let i_am_on_heap = String::from("Ferris");  
  |     ----- move occurs because `i_am_on_heap` has type `std::string::String`, which does not implement the `Copy` trait  
9 |     print_function(i_am_on_heap);  
  |                   ----- value moved here  
10 |  
11 |     let me_too = i_am_on_heap;  
  |                   ^^^^^^^^^^^^^ value used here after move
```

note: consider changing this parameter type in function `print_function` to borrow instead if owning the value isn't necessary
--> src\ownership\ownership.rs:15:25

```
15 | fn print_function(name: String) {  
  |     ----- ^^^^^ this parameter takes ownership of the value  
  |     |  
  |     in this function  
help: consider cloning the value if the performance cost is acceptable  
9 |     print_function(i_am_on_heap.clone());  
  |                   ++++++
```

단점 (싫어하는 이유)

1. 엄격한 타입

- 참조자(References)와 빌림(Borrowing)

2. 생태계와 라이브러리

- 메이저한 언어(Java, C++, Python ..)에 비해 정보 부족

3. 비동기 프로그래밍은 어렵다

- 비동기에 특화되어있지만 비동기는 어렵다
- Tokio, QUIC, WebFlux 등

Programming, scripting, and markup languages

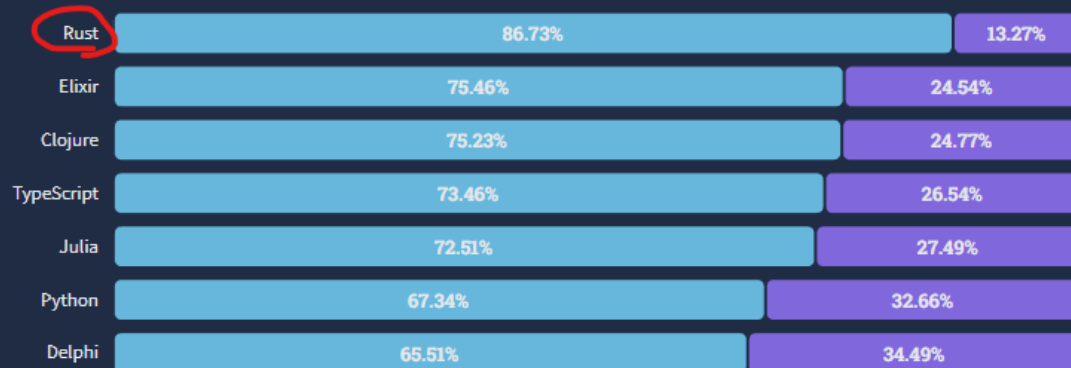
Rust is on its seventh year as the most loved language with 87% of developers saying they want to continue using it.

Rust also ties with Python as the most wanted technology with TypeScript running a close second.

Loved vs. Dreaded

Want

71,467 responses



참조자(References)

1. `&` 기호가 참조자를 의미
2. **소유권**(Ownership)을 넘기지 않고 참조 가능
3. 즉, 소유권이 없음 (스코프 밖에서 메모리 반납되지 않음)
4. 댕글링 참조자(Dangling References)가 되지 않도록 보장

```
let s1 = String::from("hello");

let len = calculate_length(&s1);

fn calculate_length(s: &String) -> usize { // s는 String의 참조자입니다
    s.len()
} // 여기서 s는 스코프 밖으로 벗어났습니다. 하지만 가리키고 있는 값에 대한 소유권이 없기
// 때문에, 아무런 일도 발생하지 않습니다. (메모리가 반납되지 않음)
```

빌림(Borrowing)

1. 함수의 파라미터로 참조자를 만드는 것을 **빌림**이라고 함
2. 빌리고 용무가 끝나면 돌려주어야함
3. 빌린 값은 고칠 수 없음 (가변 참조자는 가능)

```
fn calculate_length(s: &String) -> usize {  
    s.len()  
}
```

빌림(Borrowing)

- 변수가 불변인 것 처럼 참조자도 불변

```
fn main() {  
    let s = String::from("hello");  
  
    change(&s); // 가변 참조자: change(&mut s);  
}  
  
fn change(some_string: &String) { // 가변 참조자: fn change(some_string: &mut String)  
    some_string.push_str(", world");  
}
```

error: cannot borrow immutable borrowed content `*some_string` as mutable

--> error.rs:8:5

```
8 | | some_string.push_str(", world");  
  | | ^^^^^^^^^^^^^^^
```

가변 참조자(Mutable References, data race 방지)

1. 어떤 변수에 대해 실제 사용되는 읽기 전용 참조는 여러 개 존재할 수 있다.
2. 어떤 변수에 대해 실제 사용되는 변경 가능 참조는 단 한 개만 존재할 수 있다.
3. 어떤 변수에 대해 실제 사용되는 변경 가능 참조와, 실제 사용되는 읽기 전용 참조는 동시에 존재할 수 없다.

```
let mut s = String::from("hello");
```

```
let r1 = &mut s;
```

```
let r2 = &mut s;
```

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
```

```
--> borrow_twice.rs:5:19
```

```
4 |         let r1 = &mut s;
   |                   - first mutable borrow occurs here
5 |         let r2 = &mut s;
   |                   ^ second mutable borrow occurs here
6 |     }
   |     - first borrow ends here
```

match

- switch와 비슷하지만, match는 모든 케이스를 표현해야함 (안전성)
- `if`, `else if`, `else` 를 가독성 있게 변경

match

```
fn find_prime_number(number: u32) {  
    match number {  
        1 => println!("{}", number),  
  
        2 | 3 | 5 => println!("{}", number),  
  
        4..=10 => println!("{}", number),  
  
        n if n % 2 == 0 => println!("{}", number),  
  
        _ => println!("{}", number),  
    }  
}  
  
fn main() {  
    find_prime_number(5); // 5 is a prime number!  
}
```

match (enum)

```
#[allow(dead_code)]
pub enum Weather {
    Rain,
    Snowing,
    Foggy,
    Earthquake,
    Typhoon
}

impl Weather {
    pub fn weather_forecasting_stone(&self) -> &str {
        match self {
            Self::Rain => "Stone is wet",
            Self::Snowing => "White on top",
            Self::Foggy => "Can't see stone",
            Self::Earthquake => "Stone jumping up & down",
            Self::Typhoon => "Stone gone"
        }
    }
}

fn main() {
    println!("{:?}", weather_forecasting_stone(&Weather::Earthquake))
}
```

match (tuple)

```
fn serve_coffee(coffee: bool, ice: bool, water: bool) -> &str {  
    match (coffee, ice, water) {  
        (false, _, _) => "차가운 물만 드세요",  
        (_, false, _) => "따뜻한 아메리카노를 드세요",  
        (_, _, false) => "아이스 에스프레소를 드세요",  
        _ => "아이스 아메리카노를 드세요",  
    }  
}  
  
fn main() {  
    let coffee = true;  
    let ice = false;  
    let water = true;  
  
    let drink = serve_coffee(coffee, ice, water);  
    println!("Serve: {}", drink); // Serve: 따뜻한 아메리카노를 드세요  
}
```

result (bool)

- 실패할 가능성이 있는 값을 반환하는 Generic enum
- 파일을 읽고 실패(없는 경우)할 경우 파일을 생성하는 경우

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

```
fn main() {  
    let f = File::open("hello.txt");  
}
```

error[E0308]: mismatched types

--> src/main.rs:4:18

```
4 |         let f: u32 = File::open("hello.txt");  
    |                   ^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found enum
```

`std::result::Result`

= note: expected type `u32`

= note: found type `std::result::Result<std::fs::File, std::io::Error>`

result (파일 읽기)

- 오류 수정 후

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

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
```

result

- emacs의 rust 코드

```
fn op_system_memory_info(
  state: &mut OpState,
) -> Result<Option<MemInfo>, AnyError> {
  super::check_unstable(state, "Deno.systemMemoryInfo");
  state.borrow_mut::<Permissions>().env.check_all()?;
  match sys_info::mem_info() {
    Ok(info) => Ok(Some(MemInfo {
      total: info.total,
      free: info.free,
      available: info.avail,
      buffers: info.buffers,
      cached: info.cached,
      swap_total: info.swap_total,
      swap_free: info.swap_free,
    })),
    Err(_) => Ok(None),
  }
}
```

trait

- Java의 interface와 유사
- *trait bound*
 - Generic type이 어떤 trait를 구현한 타입인지 명시해야 함
 - 사용하길 원하는 동작을 갖도록 함

trait

- vaultwarden(bitwarden)의 db 데이터를 불러오는 함수

```
use serde::{de::DeserializeOwned, Serialize};

pub trait FromDb {
    type Output;
    #[allow(clippy::wrong_self_convention)]
    fn from_db(self) -> Self::Output;
}

impl<T: FromDb> FromDb for Vec<T> where T: Send + Serialize + DeserializeOwned {
    type Output = Vec<T::Output>;
    #[allow(clippy::wrong_self_convention)]
    #[inline(always)]
    fn from_db(self) -> Self::Output {
        self.into_iter().map(FromDb::from_db).collect()
    }
}

impl<T: FromDb> FromDb for Option<T> {
    type Output = Option<T::Output>;
    #[allow(clippy::wrong_self_convention)]
    #[inline(always)]
    fn from_db(self) -> Self::Output {
        self.map(crate::FromDb::from_db)
    }
}

// Send : thread safety
// Serialize, DeserializeOwned : 직렬화, 역직렬화가 가능해야 함
```


trait

- Person mock 후 사용 예시

```
#[derive(Debug, Serialize, Deserialize)]
struct Person {
    name: String,
    age: u32,
}

impl FromDb for Person {
    type Output = Self;

    fn from_db(self) -> Self::Output {
        self
    }
}

fn main() {
    let persons: Vec<Person> = vec![
        Person { name: "John".to_string(), age: 30 },
        Person { name: "Jane".to_string(), age: 25 },
        Person { name: "Mike".to_string(), age: 40 },
    ];

    let persons_output: Vec<Person> = FromDb::from_db(persons);
    println!("{:?}", persons_output);

    let person: Option<Person> = Some(Person { name: "Alice".to_string(), age: 35 });

    let person_output: Option<Person> = FromDb::from_db(person);
    println!("{:?}", person_output);
}
```

참고자료

<https://rinthel.github.io/rust-lang-book-ko/>

<https://prev.rust-lang.org/ko-KR/faq.html>

<https://betterprogramming.pub/reading-and-writing-a-file-in-rust-47d2bc7086ac>

<https://github.com/dani-garcia/vaultwarden>

<https://github.com/emacs-ng/deno>

<https://www.techspot.com/news/97654-how-broken-elevator-led-one-most-loved-programming.html>