

HSPACE Rust 특강

Key Features of Rust

Chris Ohk

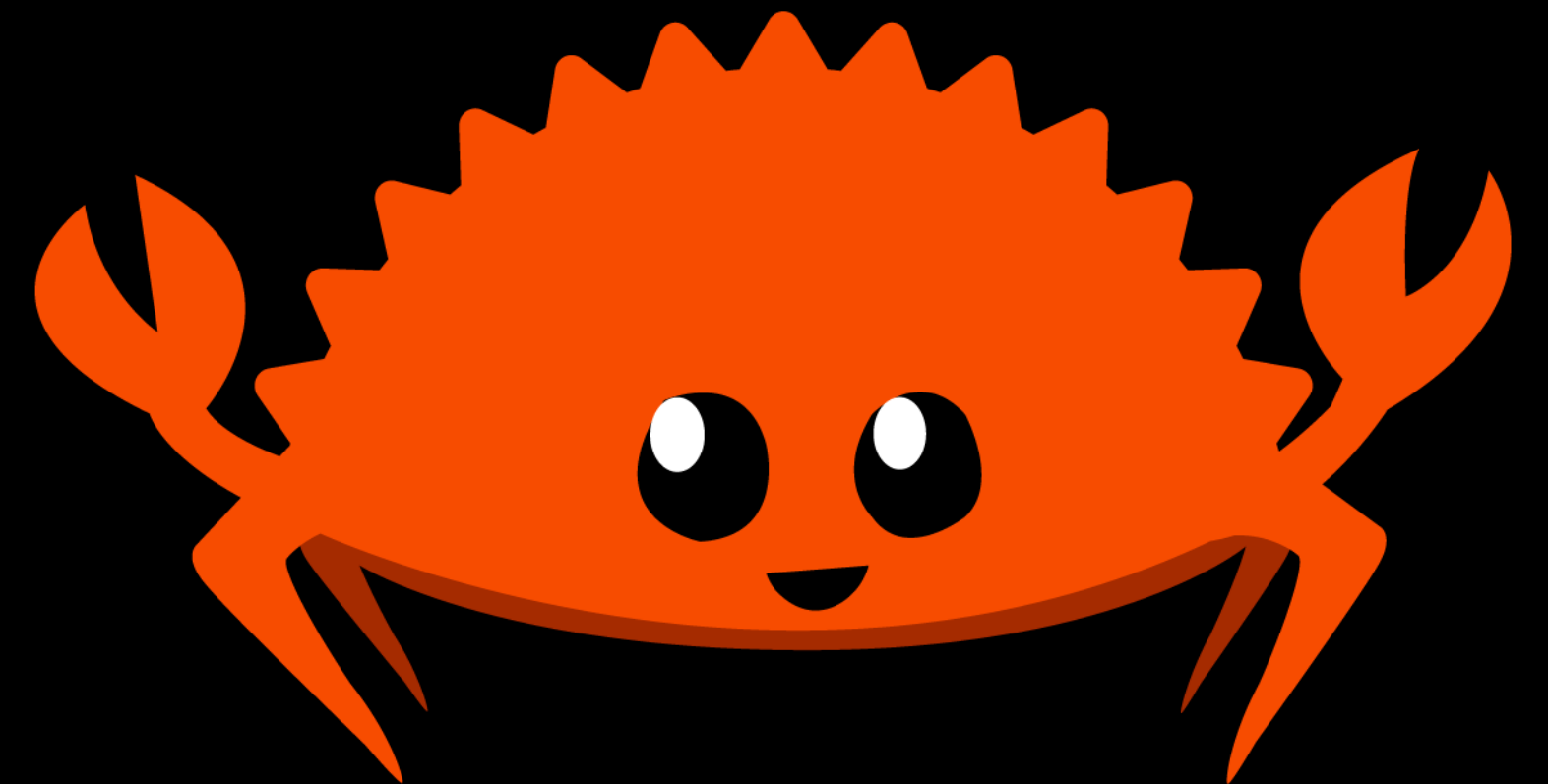
utilForever@gmail.com

- Rust 언어 소개
- Cargo : Rust의 빌드 시스템 및 패키지 매니저
- Rust는 어떻게 메모리를 안전하게 관리하는가?
- Rust에서의 OOP
- Rust는 얼마나 안전성에 진심인가?
- Rust에서 사용하는 메모리 관리는 어떻게 활용되는가?
- Rust의 꽃, 매크로

Rust란?

HSPACE Rust 특강
Key Features of Rust

- <https://www.rust-lang.org/>
- 모질라 재단에서 2010년 7월 7일 처음 발표
- 현재는 러스트 재단으로 독립해서 개발되고 있다.
- Rust 언어의 특징
 - 안전한 메모리 관리
 - 철저한 예외나 에러 관리
 - 특이한 enum 시스템
 - 트레이트
 - 하이지닉 매크로
 - 비동기 프로그래밍
 - 제네릭



- Rust의 빌드 시스템 및 패키지 매니저
- 다양한 용도로 사용할 수 있다.
 - 새 프로젝트를 만들 수 있다. (`cargo new`)
 - 프로그램을 빌드하고 실행할 수 있다. (`cargo build`, `cargo run`)
 - 프로젝트가 의존하고 있는 라이브러리를 관리할 수 있다.
 - 내장되어 있는 코드 포맷터(Formatter), 린터(Linter), 테스트 도구를 사용할 수 있다.
 - 코드 포맷터 : `cargo fmt`
 - 린터 : `cargo clippy`
 - 테스트 도구 : `cargo test`
 - 오픈 소스로 만들어진 사용자 정의 명령들을 사용할 수 있다.

Rust의 메모리 관리 방식

HSPACE Rust 특강
Key Features of Rust

- 소유권(Ownership)
- 빌림(Borrowing)
- 수명(Lifetime)

소유권(Ownership)

- 많은 언어에서는 소유자가 함수의 인자로 전달되면, 그 함수의 매개 변수로 값이 복사되거나 레퍼런스로 전달된다.

```
#include <iostream>
#include <string>

void f(std::string s)
{
    s += ", world!";
}

int main()
{
    std::string s = "Hello";

    f(s);

    // s = "Hello"
    std::cout << s << '\n';

    return 0;
}
```

소유권(Ownership)

- 하지만 Rust에서는 그 함수의 매개 변수로 값이 이동된다.

```
fn f(mut s: String) {  
    s.push_str(", world!");  
}  
  
fn main() {  
    let mut s = String::from("Hello");  
  
    // s is moved to f  
    f(s);  
  
    // Error: value used here after move  
    // println!("{s}");  
}
```

빌림(Borrowing)

- 소유권이 매번 이동되는 게 번거롭다면, 레퍼런스를 통해 잠시 소유권을 빌려줄 수 있다.
- Rust는 이때 레퍼런스 규칙을 적용한다.
 - 단 하나의 변경 가능한 레퍼런스 또는 여러개의 변경 불가능한 레퍼런스만 허용하며, 둘 중 하나만 가능하다.
 - 레퍼런스는 그 소유자보다 더 오래 살 수 없다.

```
fn main() {  
    let mut a = 10;  
    let b = &a;  
  
    // Error: cannot borrow `a` as mutable because it is also borrowed as immutable  
    {  
        let c = &mut a;  
        *c = 20;  
    }  
  
    println!("a : {a}");  
    println!("b : {b}");  
}
```


수명(Lifetime)

- Rust가 레퍼런스 규칙을 검증하는 방법.

```
#[derive(Debug)]
struct Number(i32);

// fn min(a: &Number, b: &Number) -> &Number {
//     if a.0 < b.0 {
//         a
//     } else {
//         b
//     }
// }

fn min<'a>(a: &'a Number, b: &'a Number) -> &'a Number {
    if a.0 < b.0 {
        a
    } else {
        b
    }
}

fn main() {
    let num1 = Number(5);
    let num2 = Number(24);
    let num3 = min(&num1, &num2);

    println!("num3: {num3:?}");
}
```

Rust에서의 OOP

HSPACE Rust 특강
Key Features of Rust

- Rust의 오브젝트(Object), 구조체(Struct)
- OOP의 4대 특징 : 추상화, 캡슐화, 상속, 다형성
- 트레이트(Trait)
- 트레이트 오브젝트(Trait Object)

구조체(Struct)

- Rust의 구조체는 필드(Field)만 정의하며, 메소드는 `impl` 블록에서 정의한다.

```
struct Player {  
    name: String,  
    level: i32,  
    hp: i32,  
    mp: i32,  
}  
  
impl Player {  
    fn increase_level(&mut self) {  
        self.level += 1;  
        self.hp += 10;  
        self.mp += 5;  
    }  
}
```

```
fn main() {  
    let mut player = Player {  
        name: String::from("Chris"),  
        level: 1,  
        hp: 100,  
        mp: 50,  
    };  
  
    player.increase_level();  
  
    println!("Player: {}", player.name);  
    println!("Level: {}", player.level);  
    println!("HP: {}", player.hp);  
    println!("MP: {}", player.mp);  
}
```

OOP의 4대 특징

- 추상화
 - Rust는 오브젝트의 내부 동작을 숨길 수 있다 (기본적으로 같은 모듈에서만 접근 가능하다)
 - `pub` 키워드를 사용하면 모듈 밖에서 구조체의 필드와 메소드에 접근할 수 있다.
- 캡슐화
 - 메소드의 첫번째 매개 변수로 `&self`, `&mut self` 등을 사용해 데이터와 함수를 연결한다.
- 상속, 다형성
 - 구조체는 부모 구조체로부터 필드를 상속받을 수 없다.
 - 구조체는 부모 구조체로부터 메소드를 상속받을 수 없다.

트레이잇(Trait)

- Rust에서 인터페이스나 추상 기본 클래스의 역할.
- 메소드의 집합을 구조체 데이터 타입에 연결할 수 있다.

```
trait Moveable {  
    fn move_to(&mut self, x: i32, y: i32);  
}  
  
struct Player {  
    name: String,  
    level: i32,  
    hp: i32,  
    mp: i32,  
    stamina: i32,  
}  
  
struct Pet {  
    name: String,  
    level: i32,  
    exp: i32,  
}
```

```
impl Moveable for Player {  
    fn move_to(&mut self, x: i32, y: i32) {  
        if self.stamina <= 0 {  
            println!("Not enough stamina to move");  
            return;  
        }  
  
        self.stamina -= 1;  
        println!("Moving player to ({x}, {y})");  
    }  
}  
  
impl Moveable for Pet {  
    fn move_to(&mut self, x: i32, y: i32) {  
        self.exp += 1;  
        println!("Moving pet to ({x}, {y})");  
    }  
}
```

트레잇 오브젝트(Trait Object)

HSPACE Rust 특강
Key Features of Rust

- 트레잇을 이용해 다형적 코드를 작성하는 방법.
- 가상 테이블(Virtual Table)을 통해 호출해야 할 구현을 결정한다.

```
fn main() {  
    let mut objects = vec![  
        Box::new(Player {  
            name: String::from("Chris"),  
            level: 1,  
            hp: 100,  
            mp: 50,  
            stamina: 100,  
        }) as Box<dyn Moveable>,  
        Box::new(Pet {  
            name: String::from("Dog"),  
            level: 1,  
            exp: 0,  
        }) as Box<dyn Moveable>,  
    ];  
  
    for object in objects.iter_mut() {  
        object.move_to(10, 20);  
    }  
}
```

Rust의 안전성

HSPACE Rust 특강
Key Features of Rust

- 타입 변환
- 열거체, 그리고 Option과 Result
- 패턴 매칭(Pattern Matching)
- Copy와 Clone
- 부동소숫점과 정렬

- Rust에서는 암시적인 타입 변환이 없고, `as` 키워드를 통해 명시적인 타입 변환만 가능하다.

```
fn main() {  
    let a = 10;  
    let b = 30.4;  
  
    // Error: mismatched types  
    // let c = a + b;  
  
    // Use explicit type conversion  
    let c = a as f64 + b;  
  
    println!("{c}");  
}
```


- Rust의 열거체는 데이터를 가질 수도 있고, 타입이 꼭 같을 필요도 없다.

```
enum Status {  
    Idle,  
    Run(i32),  
    Attack { damage: i32 },  
    Defend { defense: i32 },  
}  
  
fn main() {  
    let mut status = Status::Idle;  
    status = Status::Run(10);  
    status = Status::Attack { damage: 20 };  
    status = Status::Defend { defense: 5 };  
};
```

Option

- Rust에는 NULL이 존재하지 않는다.

하지만 개발을 하다 보면 NULL이 필요할 때가 있는데, 이를 위해 만들어진 타입이다.

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

```
fn main() {  
    let x: Option<i32> = Some(5);  
    let y: Option<i32> = None;  
  
    println!("{:?}", x);  
    println!("{:?}", y);  
  
    println!("{}", x.unwrap_or(0));  
  
    match y {  
        Option::Some(v) => println!("Value: {}", v),  
        Option::None => println!("No value"),  
    }  
}
```

Result

- 특정 함수의 동작 결과를 성공, 실패로 나타내기 위한 열거체 타입이다.

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

```
fn square_if_even(num: i32) -> Result<i32, String> {  
    if num % 2 == 0 {  
        Ok(num * num)  
    } else {  
        Err(String::from("Not even"))  
    }  
}  
  
fn main() {  
    let num1 = 4;  
    let num2 = 5;  
  
    match square_if_even(num1) {  
        Ok(v) => println!("Result: {v}"),  
        Err(e) => println!("Error: {e}"),  
    }  
  
    match square_if_even(num2) {  
        Ok(v) => println!("Result: {v}"),  
        Err(e) => println!("Error: {e}"),  
    }  
}
```

패턴 매칭(Pattern Matching)

- C/C++의 switch-case 문과 유사한 동작을 한다.
- Rust의 match 표현식은 값을 표현할 수 있는 모든 범위를 처리해야 한다.

```
fn main() {  
    let num = 100;  
  
    match num {  
        0 => println!("Zero");  
        1 | 3 | 5 | 7 | 9 => println!("1-digit Odd");  
        2..=8 => println!("1-digit Even");  
        num @ 10..=99 => println!("2-digit: {num}");  
        _ => println!("3-digit or more");  
    }  
}
```

Copy와 Clone

- C++에서는 얇은 복사(Shallow Copy)로 인해 댕글링 포인터 문제가 발생할 때가 있다.
이 문제를 해결하려면 깊은 복사(Deep Copy)를 해야 한다.

```
class Spreadsheet
{
public:
    Spreadsheet(int id, int rows, int cols)
        : m_id(id), m_rows(rows), m_cols(cols)
    {
        m_data = new int *[rows];

        for (int i = 0; i < rows; ++i)
        {
            m_data[i] = new int[cols];
        }
    }

    ~SpreadSheet()
    {
        for (int i = 0; i < m_rows; ++i)
        {
            delete[] m_data[i];
        }

        delete[] m_data;
    }
};
```

```
void SetCell(int row, int col, int value)
{
    m_data[row][col] = value;
}

int GetCell(int row, int col) const
{
    return m_data[row][col];
}

private:
    int m_id;
    int m_rows;
    int m_cols;
    int **m_data;
};
```

Copy와 Clone

- C++에서는 얇은 복사(Shallow Copy)로 인해 댕글링 포인터 문제가 발생할 때가 있다.
이 문제를 해결하려면 깊은 복사(Deep Copy)를 해야 한다.

```
int main()
{
    Spreadsheet sheet1(1, 10, 5);

    {
        Spreadsheet sheet2(sheet1);
        sheet1.SetCell(0, 0, 42);

        // sheet2.GetCell(0, 0) == 42
        std::cout << sheet2.GetCell(0, 0) << '\n';
    }

    // sheet1.GetCell(0, 0) == 42
    std::cout << sheet1.GetCell(0, 0) << '\n';

    return 0;
}
```

Copy와 Clone

- C++에서는 얇은 복사(Shallow Copy)로 인해 댕글링 포인터 문제가 발생할 때가 있다.
이 문제를 해결하려면 깊은 복사(Deep Copy)를 해야 한다.

```
SpreadSheet(const SpreadSheet &src)
: SpreadSheet(src.m_id, src.m_rows, src.m_cols)
{
    for (int i = 0; i < m_rows; ++i)
    {
        for (int j = 0; j < m_cols; ++j)
        {
            m_data[i][j] = src.m_data[i][j];
        }
    }
}
```



Copy와 Clone

- Rust는 얇은 복사와 깊은 복사를 위한 트레이트 Copy와 Clone을 구분하고, 얇은 복사를 했을 때 문제가 생길 수 있는 타입에 대해서는 Copy 트레이트를 구현하지 않고 Clone 트레이트만 구현해 깊은 복사만 할 수 있도록 강제한다.

```
fn hello(name: String) {  
    println!("Hello, {name}");  
}  
  
fn square(num: i32) -> i32 {  
    num * num  
}  
  
fn main() {  
    // Only implement Clone trait, not Copy  
    let name = String::from("Chris");  
    hello(name.clone());  
    hello(name);  
  
    // Implement Copy and Clone traits  
    let num = 5;  
    println!("square({num}): {}", square(num));  
    println!("square({}) : {}", num + 5, square(num + 5));  
}
```


부동소숫점과 정렬

- 다음 코드는 컴파일되지 않는다. 왜 그럴까?



```
fn main() {  
    let mut arr = vec![1.2, 4.5, 3.1, -5.7, 6.3];  
  
    arr.sort();  
  
    println!("{:?}", arr);  
}
```

- Rust에서 `sort()`를 사용하기 위해선 `Ord` 트레이트가 구현된 타입이어야 한다.
하지만 부동소숫점 타입인 `f32`와 `f64`는 `Ord` 트레이트가 구현되어 있지 않다. 왜 그럴까?

```
error[E0277]: the trait bound `{float}: Ord` is not satisfied
--> .\2 - Examples\f64_sort.rs:5:9
5   |         arr.sort();
    |         ^^^^ the trait `Ord` is not implemented for `{float}`
= help: the following other types implement trait `Ord`:
        isize
        i8
        i16
        i32
        i64
        i128
        usize
        u8
        and 4 others
note: required by a bound in `slice::<impl [T]>::sort`
--> C:\Users\utilForever\.rustup\...\rust\library\alloc\src\slice.rs:209:12
207 |         pub fn sort(&mut self)
    |         ---- required by a bound in this associated function
208 |         where
209 |             T: Ord,
    |             ^^^ required by this bound in `slice::<impl [T]>::sort`
```

- Rust는 각 연산마다 대부분 트레잇이 하나 존재한다. (예 : Add, Sub 등)
- 하지만 비교 연산은 트레잇이 2개 있다.
 - 일치 연산 : Eq, PartialEq
 - 비교 연산 : Ord, PartialOrd
- 이렇게 만든 이유는 이산수학 때 배웠던 동치 관계(Equivalence Relation) 때문이다.
 - 반사 관계(Reflexive) : 임의의 $x \in X$ 에 대하여, $x \sim x$
 - 대칭 관계(Symmetric) : 임의의 $x, y \in X$ 에 대하여, 만약 $x \sim y$ 라면 $y \sim x$
 - 추이적 관계(Transitive) : 임의의 $x, y, z \in X$ 에 대하여, 만약 $x \sim y$ 이고 $y \sim z$ 라면 $x \sim z$

- 대부분의 기본 타입은 동치 관계 조건을 모두 만족한다.
(완전 동치 관계 : Full Equivalence Relation)
- 하지만 부동소숫점은 동치 관계 조건 중 반사 관계를 만족하지 않는다.
(부분 동치 관계 : Partial Equivalence Relation)
- 그 이유는 부동소숫점 연산 과정에서 발생할 수 있는 NaN 때문이다.
이로 인해 Rust는 부동소숫점인 f32, f64 타입은 Eq, Ord 트레이트를 구현하지 않았다.

- 따라서 Rust에서 부동소숫점 타입이 저장된 컨테이너를 정렬하려면 다음과 같이 해야 한다.

```
fn main() {  
    let mut arr = vec![1.2, 4.5, 3.1, -5.7, 6.3];  
  
    // Can't use arr.sort() because f64 doesn't implement Ord  
    // arr.sort();  
  
    // Instead, use sort_by  
    arr.sort_by(|a, b| a.partial_cmp(b).unwrap());  
  
    println!("{:?}", arr);  
}
```

Rust의 메모리 관리 활용

HSPACE Rust 특강
Key Features of Rust

- 클로저(Closure)
- 동시성(Concurrency)

클로저(Closure)

- 익명 함수로, 람다 표현식(Lambda Expression)이라고 말하기도 한다.
- Rust는 클로저에 작성하는 코드에 따라 클로저의 트레잇 구현을 달리 한다.

```
fn call_twice<F>(closure: F) where F: Fn() {  
    closure();  
    closure();  
}  
  
fn main() {  
    let name = String::from("Chris");  
    let hello = || {  
        println!("Hello, {name}");  
        drop(name);  
    };  
  
    call_twice(hello);  
}
```

클로저(Closure)

- 익명 함수로, 람다 표현식(Lambda Expression)이라고 말하기도 한다.
- Rust는 클로저에 작성하는 코드에 따라 클로저의 트레잇 구현을 달리 한다.

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only implements `FnOnce`
--> .\14_closure.rs:8:17
   |
 8 |     let hello = || {
   |                  ^^ this closure implements `FnOnce`, not `Fn`
 9 |         println!("Hello, {name}");
10 |         drop(name);
   |         ---- closure is `FnOnce` because it moves the variable `name` out of its environment
...
13 |     call_twice(hello);
   |     ----- the requirement to implement `Fn` derives from here
   |     |
   |     required by a bound introduced by this call
note: required by a bound in `call_twice`
--> .\14_closure.rs:1:39
   |
 1 | fn call_twice<F>(closure: F) where F: Fn() {
   |                                     ^^^^ required by this bound in `call_twice`

error: aborting due to 1 previous error
```

For more information about this error, try `rustc --explain E0525`.

동시성(Concurrency)

- Rust의 안전성은 멀티 스레드 프로그래밍에서 빛을 발한다.

```
use std::rc::Rc;
use std::thread;

fn main() {
    let rc1 = Rc::new("Hyundai".to_string());
    let rc2 = rc1.clone();

    thread::spawn(move || {
        // Error
        rc2.clone();
    });

    rc1.clone();
}
```

동시성(Concurrency)

- Rust의 안전성은 멀티 스레드 프로그래밍에서 빛을 발한다.

```
error[E0277]: `Rc<String>` cannot be sent between threads safely
--> .\15_concurrency.rs:8:19
```

```
8 |         thread::spawn(move || {
    |                        ^-----
    |                        |
    |                        within this `{closure@.\15_concurrency.rs:8:19: 8:26}`
    |                        required by a bound introduced by this call
9 |             // Error
10 |             rc2.clone();
11 |         });
    |         ^ `Rc<String>` cannot be sent between threads safely
```

```
= help: within `{closure@.\15_concurrency.rs:8:19: 8:26}`, the trait `Send` is not implemented for `Rc<String>`,
which is required by `{closure@.\15_concurrency.rs:8:19: 8:26}: Send`
```

동시성(Concurrency)

- Rust의 안전성은 멀티 스레드 프로그래밍에서 빛을 발한다.

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

fn main() {
    let arc1 = Arc::new("Hyundai".to_string());
    let arc2 = arc1.clone();

    thread::spawn(move || {
        // No error
        let _ = arc2.clone();
    });

    let _ = arc1.clone();
}
```

매크로(Macro)

- Rust의 매크로는 함수만 가지고 할 수 있는 수준이 뛰어넘어서 다방면으로 언어를 확장하기 위한 방법을 제공한다.

```
pub use std::collections::HashMap;
pub use std::boxed::Box;
pub use std::string::ToString;

#[macro_export]
macro_rules! json {
    (null) => {
        $crate::Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        $crate::Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $( $key:tt : $value:tt ),* }) => {
        {
            let mut fields = $crate::macros::Box::new($crate::macros::HashMap::new());
            $(
                fields.insert($crate::macros::ToString::to_string($key), json!($value));
            )*
            $crate::Json::Object(fields)
        }
    };
    ($other:tt) => {
        $crate::Json::from($other)
    };
}
```

감사합니다.

utilForever@gmail.com

<https://github.com/utilForever>

Facebook, Twitter: @utilForever