Il linguaggio Rust e il suo ecosistema

Carlo Milanesi

Dalmine, Linux Day, 27 ottobre 2018

- Memory safe
- Thread safe
- Conciso
- Efficiente

Premessa: I problemi del C++

- I puntatori penzolanti (“dangling pointers”)
- L’invalidazione degli iteratori
- Le “race condition”

I puntatori penzolanti in C e C++

```
int* abc() {
    int x = 1;
    return &x;
}
```

L'invalidazione degli iteratori in C++

```cpp
// Crea un vettore di un elemento
vector<int> x { 234 };

// Prendi un iteratore sul vettore
auto iter = x.begin();

// Aggiungi un elemento al vettore
x.push_back(456);

// Accedi all'elemento referenziato dall'iteratore
cout << *iter;
```

Perché non va bene così?

- Errori in fase di esecuzione ("run-time errors")
- Errori saltuari
- Vunerabilità
- Altre soluzioni sono inefficienti (garbage collection)

Creazione di Rust

Mozilla Foundation

- 2010: Annuncio del supporto allo sviluppo di Rust
- 2015: Versione 1.0

Fondamenti di Rust

Istruzioni condizionali, cicli a condizione, funzioni

```rust
if a > 1 {
    println!("{:?}", a);
}

let mut i: i32 = 0;
while i < 10 {
    i += 1;
    println!("{:?}", i);
}

fn moltiplica(x: i32, y: i32) -> i32 {
    x * y
}
```

Fondamenti di Rust

Flusso di controllo

```
let b = 3i32;
let a = if b > 1 {
    let mut c = b - 5;
    c *= b;
    c
} else {
    1i32
};

fn maggiore(a: i32, b: i32) -> i32 {
    if a > b { a } else { b }
}
```

L'espressione `match`

```
let numero = 5u32;
let dimensione = match numero {
    0 => "nessuno",
    2 | 3 => "minuscolo",
    4...7 => "piccolo",
    8...20 => "medio",
    _ => "grande",
};
```

L'espressione match - continua

```
match numero {
    2 | 8 | 9 => {
      let mut c = 1;
      c += numero;
      println!("Il numero incrementato è {}", c);
    },
    _ => println!("Non si può incrementare")
}
```

Le strutture

```
struct Punto {
    pub x: f64,
    pub y: f64
}

let p = Punto { x: 4., y: 9.7 };
println!("La coordinata X è {}", p.x)
```

Gli enum

```
enum Forma {
    Cerchio(Punto, f64),
    Rettangolo(Punto, Punto)
}

let origine = Punto { x: 0., y: 0. };
let cerc = Forma::Cerchio(origine, 10.);
```

Il pattern-matching

```
let perimetro = match forma {
    Forma::Cerchio(_, r)
        => 2. * std::f64::consts::PI * r,
    Forma::Rettangolo(p1, p2)
        => 2. * (p2.x - p1.x).abs()
        + 2. * (p2.y - p1.y).abs(),
};
```

Il pattern-matching - continua

```
let perimetro = match forma {
    Forma::Cerchio(_, r)
        => 2. * std::f64::consts::PI * r,
    Forma::Rettangolo(
        Punto { x: x1, y: y1 },
        Punto { x: x2, y: y2 })
        => {
            2. * (x2 - x1).abs()
            + 2. * (y2 - y1).abs()
        }
};
```

I tipi generici

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

fn forse_sqrt(x: i32) -> Option<u32> {
    if x >= 0 {
        Some(sqrt(x) as u32)
    } else {
        None
    }
}
```

I tipi generici - continua

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

I metodi

```
impl Appuntito {
    // Chiamato da punto.distanza()
    fn distanza(&self) -> f32 {
        ((self.x * self.x + self.y * self.y) as f32)
            .sqrt()
    }

    // Chiamato da Appuntito::random()
    fn random() -> Punto {
        Punto {
            x: 4,
            y: 4,
        }
    }
}
```

I trait

Definizione di un trait

```rust
// Ispirati alle type classes di Haskell,
// analoghi alle interface di Java e C#.

trait Appuntito {
    fn colpisci(&self, a: &str);
    // Qui si possono anche dare
    // i corpi interi delle funzioni
}
```

I trait - continua

Implementazione di un trait

```
impl Appuntito for Punto {
    fn colpisci(&self, a: &str) {
        println!("Colpito {}", a);
    }
}
```

I trait - continua

```
// Uso di un trait

fn colpisci_per_sempre<T: Appuntito>(
    appuntito: T, a: &str)
{
    loop {
        appuntito.colpisci(a)
    }
}
```

Gestione della memoria

Gestione della memoria

Spostamenti e copie

```
let x = 5i8;
let y = x;
println!("x is {:?}", x); // y è stato copiato da x

let x = vec![1u8, 2u8, 3u8]; // Un vettore
let y = x; // il vettore è stato "spostato"
println!("y is {:?}", y);
// println!("x is {}", x); VIETATO

fn abc(x: Vec<u8>) {
}
let myvec = vec![1u8, 2u8, 3u8];
abc(myvec);
// Qui non posso più usare myvec
```

Gestione della memoria

Prestito (borrowing)

```
let x = vec![1u8, 2u8, 3u8];
let y = &x; // il valore di x è prestato a y
let c = x.clone(); // Copiato esplicitamente
println!("x is {:?}", x);
println!("y is {:?}", *y);

fn abc(x: &Vec<u8>) {
}
let myvec = vec![1u8,2u8,3u8];
abc(&myvec); // Passa un riferimento preso in prestito
// Qui posso ancora usare myvec
```

Gestione della memoria

Prestito non mutabile

```
let mut x = vec![1u8, 2u8, 3u8];
{
    let y = &x;
    println!("x is {:?}", x);
    println!("y is {:?}", y);
    // x.push(1); VIETATO
    // y.push(1); VIETATO
}
println!("x is {:?}", x);
x.push(1);
```

Gestione della memoria

Prestito mutabile

```
let mut x = vec![1u8, 2u8, 3u8];
{
    let y = &mut x;
    // println!("x is {:?}", x) // VIETATO
    // x.push(1); VIETATO
    println!("y is {:?}", y);
    y.push(1);
}
println!("x is {:?}", x);
x.push(1);
```

Gestione della memoria

Il possesso e i metodi

```
struct Poligono { punti: Vec<Punto> };
impl Poligono {
    // Sposta
    fn disegna_sposta(self) { }

    // Presta
    fn disegna_presta(&self) { }

    // Presta mutabilmente
    fn disegna_presta_mut(&mut self) { }
}
```

Gestione della memoria

Lo heap

```
let mut x: Box<u32> = Box::new(1); // Sullo heap
*x = 2;

fn abc() {
    let x = Box::new(1);
}

fn def() -> Box<u32> {
    let x = Box::new(1);
    x
}
```

Tools & libraries

Tool di build e package manager: Cargo

Repository online: https://crates.io/

Librerie mature:

- ORM (per PostgreSQL, MySQL, SQLite)
- HTTP, WebSocket
- Serializzazione (JSON, XML, CSV, ...)
- Grafica per videogiochi
- Espressioni regolari
- Crittografia

Applicabilità

Già oggi:

- Riga di comando
- Server (Web e non)
- Foreign-Function Interface

Domani anche:

- Embedded
- Device driver

Grazie dell'attenzione

Domande?