Rust: Sicuro, Veloce, Facile

LinuxDay Torino 2019

Luca Barbato

- lu_zero@gentoo.org
- lu_zero@videolan.org

Intro

Chi sono io?

- Sviluppo Gentoo, una distribuzione Linux molto flessibile e duttile.
- Contribuisco a diversi progetti sotto l'ombrellone VideoLan
- Ultimamente contribuisco a Rust e scrivo software multimediale in questo linguaggio
    - rav1e ad esempio.

Ringraziamenti

Edoardo Morandi e il resto del meetup di Torino per l'aiuto ed il supporto.

Intro

Chi siete voi?

- Quanti di voi programmano?

Intro

Chi siete voi?

- Quanti di voi programmano in Rust?

Intro

Rust in poche parole

> Un linguaggio che permette a chiunque di sviluppare software affidabile ed efficiente.
> rust-lang.org

Punti chiave

- E` un linguaggio per **tutti**.
- Per scrivere programmi **efficienti**.
    - ed **affidabili**

Solitamente ci si focalizza sull'affidabilita` ( `"Sicuro"` ) e sull'efficienza ( `"Veloce"` ), ma la prima promessa e` l'essere per *chiunque* (`"Facile"`)

Intro

Rust in altre parole (e con meno marketing)

- E` un linguaggio di programmazione di sistema che offre garanzie forti sul comportamento del codice scritto
  - Il compilatore evita una serie di tipi di errori **molto** comuni.
  - Il compilatore e` in grado di ottimizzare meglio il codice prodotto come **effetto collaterale** dell'avere maggiori informazioni.
  - Scrivere codice molto **complicato** (and esempio **multi-thread** o asincrono) diventa di conseguenza piu` **semplice**.
- E` un linguaggio in cui **ergonomia** e documentazione non sono questioni secondarie.
- E` un linguaggio che **costringe** ad essere consci di quello che si fa.
  - Il compilatore ti tiene **per mano** e spesso ti guida.

Intro

- Il linguaggio riesce ad offrire sicurezza **senza** impattare sulla velocita` di esecuzione, dato che sia una sia l'altra si basano sull'assuto che il programmatore riesca a dare **piu`** informazioni al compilatore rispetto ad altri linguaggi.
    - Il compilatore grazie a cio` riuscira` sia ad applicare piu` ottimizzazioni ed al contempo riconoscere **errori** e comportamenti **pericolosi** che in altri linguaggi possono essere individuati con maggiore difficolta` e solo con ulteriori strumenti.
- Ma questo richiede che le informazioni siano fornite.
    - Il concetto di lifetime e` piuttosto originale, cosi` come il ragionare con riferimenti e possesso in un modo abbastanza diverso rispetto a quello che succede con i linguaggi con un garbage collector puo` essere **frustrante** all'inizio.

Intro

- Sino a qui sembra che di `facile`, per chi ha gia` esperienza con altri linguaggi ci sia relativamente **poco**.
- Chi partisse con Rust come **primo** linguaggio probabilmente avrebbe da **disimparare** molto meno ed avrebbe solo i vantaggi sopra enunciati.
- Una **buona** documentazione ed una attenzione **quasi-maniacale** per i dettagli sono sufficienti a passare a questo linguaggio in maniera indolore?
    - E se non bastassero, **l'affidabilita** e **l'efficienza** dichiarate sarebbero sufficienti a farci considerare questo linguaggio un po' alieno per il nostro **prossimo** progetto?

Affidabilita`

Secondo Microsoft si`

> "As we've seen, roughly 70% of the security issues that the MSRC assigns a CVE to are memory safety issues. This means that if that software had been written in Rust, 70% of these security issues would most likely have been eliminated."
> Microsoft

Rust rende **impossibili** gli errori che han morso Microsoft per oltre i 2/3 delle volte:

- NULL-pointer dereference
- Out of bound read/write
- Use after free
- Data races

Affidabilita`

E pure per Linux se ne discute

> "With 65% of recent Linux kernel vulnerabilities being the result of memory unsafety (buffer overflows, pointers used after being freed, etc.) and not logic errors, both kernel developers and downstream users have wondered whether it's possible to use a safer language than C for kernel development."
>
> Linux Security Summit - Writing Linux Kernel Modules in Safe Rust - Geoffrey Thomas, Two Sigma Investments & Alex Gaynor, Alloy

NOTA BENE: Rust **non** risolve magicamente ogni problema, gli errori di concetto **capitano**, e vengono **corretti prima** se non hai da spendere giornate ad usare asan e/o valgrind a causa di un buffer overflow sfuggente.

Affidabilita` porta a Velocita`

Fallisci due volte in C++, riesci alla prima in Rust

> "The style component is the part of a browser that applies CSS rules to a page. This is a top-down process on the DOM tree: given the parent style, the styles of children can be calculated independently: a perfect use-case for parallel computation. By 2017, Mozilla had made two previous attempts to parallelize the style system using C++. Both had failed"
> Mozilla

Ecco un esempio perfetto di codice sicuro e ben piu` veloce: la programmazione multi-thread e` complicata e **mischiare** i possibili errori logici (che possono portare a deadlock quasi-casuali) con errori evitabili (le famose `data race`) rende molto piu` **difficoltoso** arrivare a qualcosa che funzioni bene. Con Rust hai da badare **solo alla logica**.

Velocita`

Grep e` lento quindi...

> - For both searching single files and huge directories of files, no other tool obviously stands above ripgrep in either performance or correctness.
> - ripgrep is the only tool with proper Unicode support that doesn't make you pay dearly for it.
>
> about ripgrep

- Istruzioni specifiche quali rotate, count ones, leading zeros, sono direttamente parte del linguaggio e non bisogna rinventare la ruota per supportare SIMD.
- rayon rende quasi istantaneo passare da un loop normale ad una esecuzione su un pool di thread.

Velocita` e Facilita` (di integrazione)

E perche` fermarci alla riga di comando?

> "Have you ever tried optimizing a super-slow Python application and thought: “Oh! I wish I could just write this bit it in Rust”? Well, turns out you can! We will show you how Rust is a better alternative than C to make your programs lightning fast, and how to get away with it; without your users even noticing. (see slide 113 and following)
>
> about avro-rs used from python

Facilita` (di integrazione)

Rust e` in grado di creare librerie che appaiono a tutti gli effetti come qualsiasi libreria scritta in C. E vi sono una serie di strumenti ed estensioni del build system di default, cargo, per rendere l'esperienza piacevole, o quasi:

- cargo-c: per generare librerie + header C e file pkg-config in un solo colpo.
- maturin: per creare pacchetti python e pubblicarli pure su pypi.
- wasm-pack: per creare pacchetti wasm+js e pubblicarli su npm.
- j4rs e jni-bindgen: per integrare rust e java assieme.

Facilita` (di messa in opera)

Rust puo` e viene usato per scrivere librerie ad alte performance (ad esempio da cloudflare, ma essendo un linguaggio di programmazione di sistema e` possibile usarlo anche in ambienti embedded, con sia un intero sistema operativo o componenti per fare bare metal.

Strumenti come noise explorer permettono di generare codice Rust da modelli di verifica formale, facilitando il passaggio da protocolli formalmente **corretti** a implementazioni formalmente sicure e **performanti**.

Facilita`

Facciamo un passo indietro

- Partiamo da zero e scriviamo un `hello world`.
- Vediamo quanto e` effettivamente complicato Rust con un esempio semplice
  - Che contiene incidentalmente circa meta` della sintassi e dei concetti di rust
    - Uso dei moduli ed il concetto di `path`
    - Tratti e la loro implementazione
    - Programmazione generica
    - Semantica `move` e `ownership`
  - Mancano `lifetime` esplicite, `async`, `thread`, `iteratori` ed un po' d'altro.

Hello World

```
use std::fmt;

struct Foo<T: fmt::Display> {
    a: T,
}

impl<T: fmt::Display> fmt::Display for Foo<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.a)
    }
}

fn say_hello<T: fmt::Display>(who: T) {
    println!("Hello {}", who);
}

fn main() {
    let a = Foo { a: 42 };
    say_hello("world");
    say_hello(a);
    say_hello(&a);
}
```

Hello World

```
# Installiamo rustup
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
# Usiamo cargo per creare un crate binario di esempio
$ cargo new hello_world
$ cd hello_world
# scriviam qui il nostro file.
$ vim src/main.rs
# Compila, linka ed esegue il binario
$ cargo run
```

Funzionera` tutto?

Hello World

No, borrowck ha trovato qualcosa che non gli piace!

```
error[E0382]: borrow of moved value: `a`
  --> src/main.rs:23:15
   |
20 |     let a = Foo { a: 42 };
   |         - move occurs because `a` has type `Foo<i32>`, which does not implement the `Copy` trait
21 |     say_hello("world");
22 |     say_hello(a);
   |               - value moved here
23 |     say_hello(&a);
   |               ^^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
error: Could not compile `playground`.

To learn more, run the command again with --verbose.
```

Ecco il nostro primo errore e l'amichevole compilatore a guidarci.

Intro - Hello World

```
use std::fmt;

// Effettivamente Foo non e` Copy
struct Foo<T: fmt::Display> {
    a: T,
}

impl<T: fmt::Display> fmt::Display for Foo<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.a)
    }
}

fn say_hello<T: fmt::Display>(who: T) {
    println!("Hello {}", who);
}

fn main() {
    let a = Foo { a: 42 };
    say_hello("world");
    say_hello(&a); // Foo e` un tipo move, passiamo prima il riferimento
    say_hello(a); // e poi consumiamo `a` passandolo a say_hello
}
```

Hello World

```
use std::fmt;

#[derive(Clone, Copy)] // deriviamo Clone e Copy
struct Foo<T: fmt::Display> {
    a: T,
}

impl<T: fmt::Display> fmt::Display for Foo<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.a)
    }
}

fn say_hello<T: fmt::Display>(who: T) {
    println!("Hello {}", who);
}

fn main() {
    let a = Foo { a: 42 };
    say_hello("world");
    say_hello(a); // passiamo una copia di `a`
    say_hello(&a); // a originale puo` essere ancora usato
}
```

Semplice o Facile?

- Rust sicuramente non e` semplice.

```
pub struct HashMap<K, V, S = RandomState> {
    base: base::HashMap<K, V, S>,
}

impl<K: Hash + Eq, V> HashMap<K, V, RandomState> {
    #[inline]
    pub fn new() -> HashMap<K, V, RandomState> {
        Default::default()
    }
    #[inline]
    pub fn with_capacity(capacity: usize) -> HashMap<K, V, RandomState> {
        HashMap::with_capacity_and_hasher(capacity, Default::default())
    }
}
```

Semplice o Facile?

- Ma rende incredibilmente facile fare cose complicate

```
use crossbeam::{channel, utils::thread::scope};

let (s, r) = channel::bounded(0);

scope(|scope| {
    // Spawn a thread that receives a message and then sends one.
    scope.spawn(|_| {
        r.recv().unwrap();
        s.send(2).unwrap();
    });

    // Send a message and then receive one.
    s.send(1).unwrap();
    r.recv().unwrap();
}).unwrap();
```

Semplice o Facile?

- La complessita` viene nascosta molto bene

```
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter() // <-- la map viene eseguita in parallelo da un pool di worker
         .map(|&i| i * i)
         .sum()
}
```

E grazie al compilatore le astrazioni in media **non** si pagano in tempo di esecuzione, **ma** i tempi di compilazione possono soffrirne molto.

Semplice o facile?

- Gli strumenti per essere produttivi dal primo minuto sono tutti disponibili:

```
$ rustup doc # tutta la documentazione della libreria standard e` disponibile anche offline
$ cargo fmt # per avere uno stile uniforme
$ cargo clippy # per scrivere codice piu` idiomatico
$ rustup component add rls rust-analysis rust-src # per avere il completamento automatico su diversi IDE
$ rustup component add miri # per chi vuole una analisi piu` approfondita del suo codice
```

- E ve ne sono molti altri

```
$ cargo install flamegraph # per chi vuole fare benchmark
$ cargo install cargo-asm # per chi vuole vedere cosa ha fatto il compilatore effettivamente
$ cargo install cargo-bloat # per chi vuole ottimizzare anche la dimensione dei binari
$ cargo install bolero-cargo # per chi vuole fare property testing senza faticare troppo
```

Conclusione

Sono ben 3 anni che parlo di Rust al LinuxDay:

- Nel **2017** ho raccontato Rust, spiegando perche` **non** lo si dovesse usare per riscrivere il mondo.
- Nel **2018** ho illustrato come mai Rust sia stato scelto per scrivere rav1e.
- **Oggi** vi ho raccontato perche` ora ci sono poche ragioni per non usarlo.

Domande?