04-04.md

1. Lezione del 04/04 - Rust e Polimorfismo

Come già detto Rust non è propriamente un linguaggio ad oggetti in quanto non implementa un vero e proprio meccanismo di ereditarietà. Per via di ciò il polimorfismo qui è inteso come il dare funzionalità comuni a strutture diverse attraverso i tratti.

Il meccanismo dei tratti se vogliamo è ispirato al concetto delle interfacce in altri linguaggi. Su di essi il linguaggi implementa tutte le funzionalità di base e non solo.

Ad esempio in Rust possiamo stampare una variabile in due modi differenti, ossia facendo interpolazione dei caratteri `{}` all'interno di una stringa o dei caratteri `{:?}`, con la differenza che nel secondo caso stiamo chiedendo la stampa in modalità Debug del dato. Questo comportamento è definito secondo un tratto.

La sintassi è fondamnetalmente simile a quanto visto in altri linguaggi, ma il principio è avere una funzione che definisce il tratto.

```
// Definizione del tratto

trait Something {
    fn someOperation(&mut self) -> SomeResult;
}

...

// Implementazione

impl Something for SomeType {
    ....
}
```

Uno delle inconvenienze tipice della programmazione è la scrittura frequente di pezzi di codice triviali. Per alcuni tratti ed alcune casistiche, Rust permette di ovviare a tale cosa.

Quando si vuole utilizzare questa caratteristica invece iniziano i mal di testa. Il linguaggio offre quetsa possibilità in due modi. Ad esempio:

```
trait Default {
    fn default() -> Self;
}
```

Questa funzionalità non ha parametro quindi non la posso applicare ad una struct, e pertanto va richiamato con la sintassi `Default::default()` e pertanto è detta riflessiva, come fosse un metodo statico di altri linguaggi, ma possiamo anche dire che è possibile chiamarlo facendo riferimento al tipo. Questo significa che è possibile definire prima il tipo da cui il compilatore andrà a recuperare il metodo per tale tipo, se implementato

```
fn main() {
    let zero: i32 = Default::default(); // i32::default() richiamato implicitamente
    let zero_again = i32::default();    // i32::default() richiamato esplicitamente
}
```

Tecnicamanete il parametro self sarebbe il nome richiamato all'interno della funzione e sarebbe `Self` con la S maiuscola, con tutte le variazioni desiderabili di `&Self` e `&mut Self`.

```
trait T2 {
    fn takes_self(self);
    fn takes_self_again(self: Self);    // Equivalenti
}
```

In Rust posso anche definire un tipo come secondario, in modo da essere richimato con più facilità. Questo concetto è meglio assimilato tramite un esempio, quindi facciamo finta di star implementando una struttura `Containter` che vuole implementare un tratto detto `Contains`:

```
trait Contains {
    type A;
    type B;
    ...
}
```

Le funzioni che vogliono utilizzare il tratto Contains una volta che esso è stato implemntato per una struttura ora non dovranno più esplicitarlo. Per confronto, proviamo ad implemenare la funzione `contains` prima senza i tipi associati e poi con.

Senza:

```
trait Contains<A, B> {
    fn contains(&self, _: &A, _: &B) -> bool;
}

impl Contains<i32, i32> for Container {
    fn contains(&self, _: &A, _: &B) -> bool {
        ...
    }
}
```

Ora vediamo la sintassi con il tipo associato:

```
trait Contains {
    type A;
    type B;

    fn contains(&self, _: &Sefl::A, _: &Self::B) -> bool;
}

impl Contains for Container {
    type A = i32;
    type B = i32;
    fn contains(&self, _: &i32, _: &i32) -> bool {
        ...
    }
}
```

L'uso dei tipi associati consente anche di non dover specificare affatto i tipi cui si fa riferimento dopo averli definiti.

```
fn difference<A, B, C>(container: &C) -> i32 where C: Contains<A, B> {...} // Senza tip

fn difference<C: Contains>(container: &C) -> i32 { ... } // Con tipi associati
```

Come visto in altri linguaggi è possibile, in fase di definizione di un tratto, definire delle implementazioni di default delle funzioni che vanno a richiedere, in modo da lasciarle inalterate qualora non sia strettamente necessario la loro ridefinizione.

```
trait Tratto {
    fn f(&self) {
        println!{"Default action, not implemented"}
```

```
    }
}

struct Struttura {
    ...
}

impl Tratto for Struttura;

let struttura = Struttura;
struttura.f(); // "Default action, not implemented"
```

Sebbene Rust non abbia una linea di ereditarietà delle strutture, esso dispone di una linea ereditaria dei tratti. Posso cioè definire per un tratto un suo sottotratto, che rende automaticamente il tratto originale un supertratto. Questo crea però delle ambiguità nel caso delle chiamate a funzioni, che se vengono re-implemenate hanno bisogno di esplicitare quale delle due si va a chiamare.

```
trait Supertratto {
    fn f(&self) { println!{"Supertratto"}}
}

trait Sottotratto : Supertratto {
    fn f(&self) { println!{"Sottotratto"}}
}

struct Struttura {
    fn new() -> Self {
        Struttura
    }
}

impl Supertratto for Struttura {} // Impl. di Default
impl Sottotratto for Struttura {} // Impl. di Default

fn main() {
    let struttura = Struttura::new();
    Sottotratto::f(&struttura); // "Sottotratto"
    Supertratto::f(&struttura); // "Supertratto"
}
```

Come già accennato altre volte, grazie ai tratti possiamo richiedere di non conoscere esplicitamente il tipo di dato su cui una funzione va a lavorare ma a richiedere al compilatore che esso implementi i tratti necessari. Per fare ciò si fa uso della parola chiave `dyn` che accetta un tratto o una combinazione di tratti.

```
trait Stampa {
    fn stampa(&self);
}

impl Stampa for Struttura {
    fn stampa(&self) {
        println!{"Stampa struttura"};
    }
}

fn process( val: &dyn Stampa) {
    val.stampa();
}
```

```
fn due_tratti_richiesti(val: &dyn Stampa+Clone) {
    val.stampa();
    val.clone();
}
```

Per far si che la cosa funzioni, la referenza a `dyn` è un fat pointer che include una referenza al dato, più nello specifico un'istanza della struttura ovvero quella su cui si chiama il tipo, e l'altro a una tabella virtuale che contiene i puntatori alle funzioni da richiamare. A runtime, quando si richiama la funzione sul tipo $dyn Tratto$, si consulta la tabella per ottenere il puntatore e poi tale funzione è eseguita. Potremmo dire che, come in C, il puntatore al dato è un puntatore generico, come fosse un `void *`, e come tale porta delle leggere inefficienze che il compilatore non puo' superare in quanto questo funzionamento è gestito a runtime.

In Rust non esiste l'overloading degli operatori. Di fatto, quando chiamiamo Rust in linugaggio per tratti, è anche dovuto al fatto che persino la ridefinizione degli operatori di somma, sottrazione e quant'altro è gestito tramite i tratti.

Per ridefinire l'operazione di somma di una struttura:

```
impl Add for Struttura {
    type Output = Struttura;

    fn add(self, other: Struttura) -> Struttura {
        // La nostra operazione di somma
    }
}
```

La cosa interessante di modo di operare è che è persino possibile ridefinire tali operazioni per tipi di dato etorgenei. Anche qui, un esempio chiarisce subito il concetto:

```
use std::ops::Add;

struct Punto {
    x: i32,
    y: i32,
}

impl Add<i32> for Punto {
    fn add(&self, other: i32){
        Punto {
            x: self.x + other,
            y: self.y + other,
        }
    }
}

...

let p = Punto(3, 4);
println!("{}", p); // { x: 3, y: 4 }
println!("{}", p + 3); // { x: 6, y: 7 }
```

Notiamo che in Rust esistono tratti per l'operatore di equalità, `Eq` e `PartialEq`. `PartialEq` richiede che siano garantite dall'implementazione sia la proprietà simmetrica che transitiva. Per simmetria si intende che se `A == B` allora `B == A`, mentre per transitività se `A == B` e `B == C` allora `A == C`. Il tratto `Eq` impone anche la proprietà di essere uguali a se stessi, e puo' sembrare una banalità ma non è così. Ad esempio, per i tratti floating point non è definito `Eq`, in quanto si ha che `NaN != NaN`. `PartialEq` è definito come un supertratto di `Eq`.

Due tratti importanti ma che raramente andremo a implementare da noi sono `Clone` e `Copy`. Nello specifico, `Clone` è quasi sempre superfluo da modificare oltre quanto fornitoci dalla macro `#[derive(Clone)]`, mentre `Copy` non è un tratto implementabile in altro modo.

I due tratti sono simili con delle sottigliezze a differenziarli. `Clone` è funzionalmente quello che noi chiederemmo da un operatore di copia, ovvero copia dei dati contenuti in una struttura ma non dei riferimenti. `Copy` dall'altro lato, non esegue copia dei dati ma dei riferimenti.

Spieghiamo nuovamente questo concetto con un esempio. Supponiamo una basilare struttura di questo tipo:

```
struct ContenitoreVec {
    vec: Vec<i32>,
}
```

La differenza tra `Copy` e `Clone` è la seguente. Con `Clone` si allocherebbe lo spazio per una nuova struttura, e si eseguirebbe la copia dei dati all'interno, nel senso che si creerebbe un altro vettore il cui contenuto è una copia dei dati dell'originale, ma in due indirizzi diversi. La `Copy` invece è definita come copia byte a byte, ovvero si copierebbe il riferimento al vec originale, con le due strutture che puntano allo stesso indirizzo e allo stesso `Vec`.

Il tratto `Drop` invece permette di definire il tratto `drop(&mut self)` che è analogo a quello che chiameremmo distruttore. È in mutua esclusione col tratto `Copy`, vale a dire che se implementiamo il tratto `Drop` per un tipo non possiamo definirne anche il tratto `Drop` e viceversa.

Potremmo aver bisogno di definire un sistema di indicizzazione. Significa richiamare le parentesi quadre come faremmo ad esempio su un vettore (`[i]`) e definire il comportamento desiderato. In Rust se ne hanno due tipi, quello "normale" e quello "mutabile". Questo perchè in lettura vogliamo ottenere un riferimento non mutabile ed in scrittura un riferimento mutabile.

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self index: Idx) -> &Self::Output;
}
```

`Sized` è un tratto implicito che indica che la dimensione occupata dalla struct è nota in fase di compilazione. `?Sized` rimuove questo vincolo.

In Rust è possibile esplicitare l'accesso per riferimento, ma in lettura questo è poco utile. Per poter modificare l'elemento, dovremmo richiedere il riferimento mutabile.

```
let vec = vec![1, 2, 3, 4, 5];
let nref : &i32 = &vec[0]; // Riferimento non mutabile
let mut mref : &i32 = &mut vec[0]; // Riferimento mutabile
```

Anche la deferenziazione è un tratto in Rust. Questo potrà sembrare strano ma non lo è, poiché la mia struttura potrebbe contenere un dato sotto forma di puntatore e voglio definire come accedervi senza esporlo pubblicamante. Siccome in Rust si ha una netta distinzione tra cosa è mutabile e cosa non lo è, abbiamo due tratti di dereferenziazione, `Deref` e `DerefMut`.