Rust::tipi_composti

Sommario

Struct

Spesso e nei contesti più svariati occorre mantenere uniti insiemi di informazioni eterogenee. A questo punto ci si chiede perché non utilizzare le *tuple*. Queste seppure permettono ugualmente di racchiudere informazioni eterogenee, nascondono la semantica dei campi di cui sono costituite.
Una `struct` invece è costituita da un **insieme di campi** il cui *nome* e il cui *tipo* sono definiti dal programmatore. Il seguente è un esempio:

```
struct MioTipo{
    nome_alunno: String,
    eta: i32,
    matricola: usize
}
```

Si usano per le struct delle convenzioni sui nomi e in particolare:

- Il nome del tipo (nome della struct) viene espresso in `CamelCase`
- Il **nome dei campi** viene espresso invece in `snake_case` .

Dichiarazione e uso di una struct

```
#[derive(Debug)]
struct MioTipo{
    nome: String,
    eta: usize
}
fn main{
    let my_struct = MioTipo{nome: "Carlo".to_string, eta: 34};
    println!("Nome {:?} Eta {:?}", my_struct.nome, my_struct.eta);
}
```

Il codice sopra mostra come **dichiarare** e **inizializzare** una struct, i suoi campi sono accessibili tramite la *dot-notation* (`variabile.campo`).
E' importante notare che ogni `struct` introduce un **nuovo tipo** il cui nome coincide con il nome della struct stessa.
Il Rust consente di avere una certa flessibilità permettendo di:

- Dichiarare delle struct simili a delle tuple nel modo seguente

```
struct Data(i32, i32, i32);
let my_Data=Data(24,7,2000);
```

- Dichiarare **struct vuote** che hanno un tipo analogo a `()` nel modo seguente

```
struct StructVuota;
let my_vec:Vec<StructVuota> = Vec::new();
```

Visibilità

Poiché una struct (un Tipo) potrebbe essere utilizzata in una porzione di codice diversa dal modulo in cui essa è stata dichiarata, il linguaggio Rust fornisce un *modificatore di visibilità* per regolare questo aspetto.
Nello specifico, di base tutti i campi di una struct sono **privati** cioè accessibili solo al codice contenuto nel modulo in cui la struttura è stata definita e ai suoi sottomoduli. Per rendere invece accessibile all'esterno la *struct definita* e i *suoi campi* bisogna anteporre ad ognuno la parola chiave `pub` . In altri

linguaggi orientati agli oggetti come il `Java` la parola 'privato' assume un significato diverso! In Rust non esistono le classi, per cui le regole di visibilità sono riferite *solo* ai moduli.

Moduli

Abbiamo citato più volte il concetto di **modulo** ne introduciamo l'uso in questo paragrafo. Di base i moduli sono utilizzabili per dividere il codice in *porzioni indipendenti*. I moduli per questa loro caratteristica permettono di implementare il meccanismo di *information hiding*. Per essere però efficace, è utile associare dei *comportamenti* alla struct.
Mentre in altri linguaggi, quali `Java` e `C++` definizione della struttura dati e implementazione risiedono in un unica struttura che è la *classe*, in `Rust` , l'implementazione risiede in un blocco separato introdotto da `impl` .

Il segmento di codice seguente riporta la definizione e l'utilizzo di un modulo.

Definizione di un modulo

```
module MioModulo{
    pub struct MiaStruct{
        campo1: i32
    };
    pub fn funzione_stampa(){
        println!("sono una funzione");
    }
}
```

Per essere utilizzate, la struct e la funzione, hanno bisogno del modificatore `pub` a cui prima abbiamo fatto cenno. Questo le rende pubbliche per l'utilizzo in porzioni di codice che non si trovano nello stesso modulo in cui la struttura è stata definita.

Utilizzo di un modulo

```
use MioModulo::MiaStruct;
use MioModulo::funzione_stampa;

fn main(){
    let a = MiaStruct(campo1: 15);      //Dichiarazione
    //Utilizzo della funzione del modulo
    funzione_stampa();
}
```

Metodi

I metodi collegati ad un certo tipo si possono definire all'interno di un blocco di codice racchiuso tra parentesi graffe e introdotto da `impl <NomeTipo>` .
Se all'interno di questo blocco le funzioni hanno come primo parametro uno tra `self` , `&self` , `&mut self` quella funzione diventa un **metodo** per le istanze di quel particolare tipo.
Anche in `Rust` si è soliti utilizzare la notazione *instance.method()*, dove:

- *instance* è un'istanza del tipo in questione, chiamata anche **ricevitore** del metodo;
- *method* costituisce il nome della funzione.

L'implementazione del metodo ha accesso ai dati della struttura tramite `self` e le sue varianti che mi permettono di specificare **come** il metodo tratta l'istanza della struttura che riceve come parametro.

Il **primo parametro** di un metodo definisce il *livello di accesso* che il codice del metodo ha sul ricevitore, in particolare:

- `self` indica che il ricevitore viene passato per **movimento** consumando il valore della variabile. E' la forma compatta di `self: Self` ;
- `&self` , in questo caso il ricevitore viene passato per **riferimento condiviso**, il dato della struttura NON viene consumato; è una forma compatta per indicare `&self: &Self` ;
- `&mut self` , ricevitore passato per **riferimento mutabile**, anche in questo caso è la forma compatta per `&mut self: &mut Self` .
  A questo punto uno si potrebbe domandare: *qual è la differenza tra `self` e `Self` ?* E' abbastanza semplice. Il primo rappresenta l'istanza (o il ricevitore) il secondo identifica invece il **tipo** per cui si sta implementando il metodo.

Quando invece un metodo non ha come primo parametro `self` , un suo riferimento o riferimento mutabile, si cotruisce quello che in linguaggi come Java, è rivestito dal ruolo di metodi statici e costruttori.

Riassumiamo ora con un esempio tutti i concetti che abbiamo introdotto.

```rust
//Definizione struttura
#[derive(Debug)]
struct Punto{
    x: 32,
    y: i32,
    z: i32
}
//Implementazione struttura (comportamenti associati)
impl Punto{
    fn get_x(&self)->i32{
        self.x;
    }
    fn add_one(&mut self){
        self.x+1;
        self.y+1;
        self.z+1;
    }
    fn transform(self)->Self{
        Self{x:self.x+3, y:self.y+3, z:self.z+3}
    }
}
//Utilizzo del tipo nuovo
fn main{
    let A = Punto{x:0, y:0, z:0};
    //meotodo che usa A come riferimento condiviso
    let x = A.get_x;
    //metodo che usa A come riferimento mutabile
    A.add_one();
    //metodo che usa il ricevitore per movimento
    let B = A.transform();
    //da qui in poi A risulta 'uninitialized'
}
```

Costruttori

In `Rust` non esiste il concetto di **costruttore**, un'istanza di un certo tipo può essere creata in qualsiasi punto del codice. Tuttavia per evitare *duplicazioni del codice*, nelle implementazioni dei tipi vengono definite delle *funzioni di inizializzazione* del tipo `fn new()->Self{...}`.
Inoltre, dato che in `Rust` non esiste il concetto di overloading delle funzioni, spesso si ricorre a più funzioni di inizializzazione diverse del tipo `fn with_details(...)->Self`

```
impl Punto{
    ...
    //Funzione di inizializzazione
    fn new()->Self{
        Self{x:0,y:0,z:0}
    }
    //Inizializzazione alternativa
    fn with_value(X: i32, Y:i32, Z:i32)->Self{
        Self{x:X,Y:y,Z:z}
    }
    ...
}
//Utilizzo
fn main(){
    let var=Punto::new();m
    let var_b=Punto::with_value(10,10,10);
    ...
}
```

Distruttori

Il Rust gestisce il rilascio di risorse acquisite da un'istanza utilizzando il tratto `Drop` , questo ha associato una funzione `drop(&mut self)->()` che viene chiamato quando una variabile di un certo tipo **esce dallo scope sintattico**. Si può inoltre **forzare il rilascio** delle risorse utilizzando la funzione `drop(mio_oggetto)` , che rilasciando le risorse associate, determina l'uscita della variabile dallo scope sintattico.

Il Paradigma RAII

Nel linguaggio C++ esiste una funzione chiamata **distruttore** che viene introdotta da `~<NomeTipo>` , similmente il compilatore chiama questo metodo quando l'istanza della classe esce dallo scope sintattico o quando si chiama `delete` .
La presenza del distruttore *abilita* un particolare paradigma detto **RAII**(Resource Acquisition is initialization): *poiché il distruttore viene chiamato nel momento in cui una variabile esce dallo scope sintattico, allora la coppia* ***costruttore-distruttore*** *può essere utilizzata affinché determinate operazioni siano fatte in una porzione di codice in cui ci sia una variabile appositamente dichiarata*.
Il **paradigma RAII** può essere espresso in sintesi nel modo seguente:

- Le **risorse** sono incapsulate all'interno di una struttura in cui:
    - il costruttore acquisisce le risorse e lancia un'eccezione nel caso in cui questo non sia possibile;

    - il distruttore ha il compito di rilasciare solo le risorse senza sollevare alcuna eccezione.
- Una classe che è RAII compatibile:
    - Ha una **gestione automatica** del tempo di vita, oppure
    - E' legato al tempo di vita di un altro oggetto (ad esempio è una parte di esso).

> In realtà il nome del paradigma RAII sarebbe dovuto essere **Resource Finalization is Destruction**(RFID) utilizzato però già in altri ambiti.

In ambiente `Rust` per implementare correttamente il paradigma ci deve essere la garanzia che ogni volta che venga invocato `Drop` ci sia stata una sola costruzione dell'oggetto.
Proprio per questo motivo, il tratto `Copy` è mutuamente esclusivo con il tratto `Drop` .

Metodi statici

Sono definite anche *funzioni associate* e si possono introdurre nell'implementazione del tipo semplicemente non mettendo come primo parametro `self` . Come in tutti i linguaggi di programmazione, sono funzione **legate al tipo** più che all'istanza del tipo.
Mostriamo un esempio di metodo statico per il tipo `Punto` prima definito

```
impl Punto{
    ...
    fn origin()->Self{
        Self{x:0, y:0, z:0}
    }
    ...
}

fn main(){
    let origine=Punto::origin();
}
```

Enum

`enum` e clusola `match`

Enumerazioni generiche: `Option<T>` e `Result<T,E>`