05-09.md

1. Lezione del 09/05 – Iteratori

Iniziamo adesso a discutere degli iteratori. Essi non sono altro che una struttura che avvolge un contenitore e mantiene una serie di informazioni aggiuntive sul suo stato.

Come ormai da prassi nel linguaggio, tale funzionalità è raggiunta mediante i tratti e offre 3 modalità di accesso. Iniziamo col caso in cui volessimo iterare su di una collezione già esistente, prendiamo ad esempio un `Vec`.

Suppoinendo di avere un vettore `vec` di tipo `Vec<T>`, possiamo iterare su esso come: - vec.iter() - iteratore di `&T`, quindi borrow non mutabili e dunque utili in lettura. Non consuma l'iteratore quindi possiamo richiamarlo più volte - vec.iter*mut() - iteratore di `&mut T`, uguale al precedente ma che permette la modifica del dato nella collezione, permette quindi la modifica dei dati e puo' essere riutilizzato - vec.into*iter() - caso più particolare, iteratore di `T` e quindi causa di movimento e trasferimento di proprietà. Il dato puo' essere quindi modificato come con `iter_mut` ma una volta usato il contenitore iniziale non esiste più.

In generale il linguaggio espande il nostro codice scritto in maniera semplice e fruibile non modo più complesso. Una semplice sequenza come:

```
for i in 0..10 {
    println!("{}", i);
}
```

Viene espansa a:

```
let iter = (0..10).into_iter();
loop {
    let i = iter.next();
    match i {
        Some(i) => println!("{}", i),
        None => break,
    }
}
```

Per comprendere questo codice dobbiamo parlare del tratto `Iterator`. Implementare tale tratto significa definire un'unica funzione `next(&self mut)` che ritorna un `Option`, nel caso sia presente il risultato sarà `Some(prossimo_elemento)` e nel caso l'iteratore sia esausto si ritorna un `None`.

Notiamo quindi che la semantica dei `for` loop ci solleva dal dover eseguire esplicitamente. Altra caratteristica fondamnetale ci solleva anche dal dover scegliere il tipo di funzione da chiamare.

Notiamo infatti che è comparsa una chiamata a `into_iter`, che è stata scelta dal linguaggio che puo' usare una qualunque delle 3 alternative proposte. Contestualmente dall'uso che se ne fa nel ciclo `for` il compilatore aggiungerà tale chiamata, seguendo l'ordine `&T`, `&mut T` e `T`. Il compilatore sceglierà l'opzione più restrittiva che permetta comunque di raggiungere la funzionalità.

Notiamo infine che l'implementazione di `Iterator` utilizza un tipo associato:

```
struct Struttura<T> {
    dati : Collezione<T>
}

impl<T> for Struttura { ... }

impl Iterator for Struttura {
```

```
    type Item = T;
    fn next(&mut self) -> Option<T> {
        // Operazione per ottenere il prossimo elemento oppure None
    }
}
```

Anche gli iteratori sono soggetti al controllo di proprietà. Non sarà infatti possibile chiamare `iter_mut()` su una collezione non dichiarata come mutabile oppure su collezioni di cui non si possa garantire la consistenza di dato con tali operazioni. Ad esempio, non possiamo iterare con referenze mutabili su un `HashSet`, in quanto variando il dato varierebbe anche la chiave e la collezione si troverebbe poi in uno stato inconsistente.

Abbiamo parlato di iteratori in Rust come un modo di eseguire un'operazione su ogni singolo elemento di una collezione, quindi è giusto parlare anche delle collezioni stesse.

Come è imamaginabile, Rust offre di base nella sua libreria standard tutti i tipi di collezioni che si si aspetterebbe. Parliamo dunque brevemente della loro implementazione e delle casistiche in cui andrebbero usate.

Quanto studiato in altri specifici per strutture dati rimane ovviamente valido anche qui, ma dobbiamo parlare di quello che succede quando da un approccio teorico si vuole andare a considerare le prestazioni su un sitema reale.

(Image: *../imgs/20220614143929.png*)

Nella quasi totalità dei casi, tenderemo a escludere le liste, in Rust offerte dalla struttura `LinkedList`. Infatti oltre al tipo `Vec` la libreria standard offre anche il tipo `VecDeque` che è un vettore efficiente a inserire e rimuovere elementi sia in testa che in coda, in quanto realizzato a buffer circolare.

Questo sposta la sua efficienza in tutte le operazioni al pari di una lista ma, essendo comunque memorizzata in memoria contigua sullo heap, e considerando un sistema reale che quindi implementa paginazione e caching, risulterà più veloce sui benchmark.

In genere in caso di parità per l'analisi di complessità per una data operazione, `Vec` risulterà più veloce di `VecDeque` che a sua volta risulterà più veloce di `LinkedList`.

Le restanti strutture dati invece seguono effetivamente quanto visto nei corsi di strutture dati, con una differenza tra collezioni basate su mappe e collezioni basate su alberi. Infatti se si desiderasse una collezione che faccia esattamente quanto previsto da essa si dovrebbe sempre fare affidamento alla versione hash di essa (`HashSet` o `HashMap`) mentre nel caso in cui tale collezione debba essere utilizzata per acquisire un intervallo di valori, inteso come tutti gli elementi nella mappa la cui chiave sia minore di un'altra per dire, le versioni `BTreeSet` e `BTreeMap` risulteranno più performanti.

In ultimo, ogni collezione offre due funzioni che ci permettono di ottimizzare i nostri programmi sia nel tempo che nello spazio, ovvero i metodi `with_capacity` e `shrink_to_fit`.

Nel caso in cui si conosca in anticipo il numero di elementi che si andranno a inserire infatti si potrebbe creare la collezione tramite il metodo `<NomeCollezione>::with_capacity(capacity)` per allocare preventivamente quello spazio, in modo da minimizzare le riallocazioni e ottenere grossi aumenti di prestazione. Se invece se ne conosce solo un (ragionevole) limite superiore, e poi se ne utilizzino effettivamente di meno e si voglia davvero sfruttare al massimo la memoria, il metodo `shrink_to_fit` permette di ridurre le dimensioni in modo da utilizzare meno memoria possibile, ma non è detto che la capacità finale coincida col numero di elementi contenuti, in quanto potrebbe essere leggermente superiore.

Finita questa digressione sulle strutture dati passiamo ora a parlare di I/O. Che si parli di Rust o altri linguaggi ad alto livello, per esso troviamo sempre un concetto cardine per leggere e scrivere, e vale a dire il concetto di interfaccia. Questo significa che tutti i linguaggi, a partire dal C, hanno fatto uno sforzo per far si che il programmatore, indifferentemente da dove debba andare a scrivere o leggere le infromazioni, sia file, console o altro, utilizzi sempre le stesse funzioni.

Come in altri linguaggi anche qui sono presenti le funzioni per aprire un file in sola lettura o sola scrittura, comunque differenziando il caso in cui il file esista da quello in cui non esista, ed abbiamo le funzioni di seek per spostarci all'interno del file o aprirlo portandoci direttamente già alla fine di esso.

Tutte queste sono per l'appunto interfacce, che ci permettono di scrivere sequenze logiche o stringhe richiamando in realtà funzionalità di sistema che scrivono o leggono dati binari puri (come sequenze di byte).

```
use std::fs::File;
use std::io::{Write, Buffer, BufReader, BufWriter};

let path = "lines.txt";
let mut output = File::create(path).unwrap();

let input = File::open(path)?;
let buffered = BufReader::new(input);

for line in buffered.lines() {
    println!(output, "{}", line?);
}
```