05-10.md

1. Lezione del 10/05 – Puntatori

Parliamo ora di una caratteristica probabilmente nuova e che è propria dei linguaggi alto livello per la programmazione basso livello, vara a dire i puntatori intelligenti.

In C un puntatore è niente più un indirizzo di memoria associato al tipo di dato puntato. L'intera logica della sua gestione e del suo utilizzo è demandato al programmatore, e questo tende a creare dei problemi quando voglio condividere questo dato in più parti del mio programma.

Non potendo infatti tener traccia, per lo meno non in modo conveniente, di quante volte abbia condiviso questo stato e del se quel dato è ancora valido, la probabilità di incappare in errore è alta.

Già il C++, pur essendo solo un superset del C, ha quindi introdotto dei tipi di dato costruiti su essi, detti puntatori intelligenti, per consentire la più facile gestione della memoria dinamica, seppur non avendo il C++ a differenza di Rust un borrow checker non riesce a implementare tutte le funzionalità di Rust, almeno non in maniera idiomatica.

Quello che voglio ottenere con i puntatori intelligenti è un comportamento per cui tutti i controlli che in C sono ottenuti tramite frequenti e manuali controlli, come l'assicurarsi che un dato sia valido prima di accedervi o la liberazione della memoria alla fine dell'esecuzione di un programma, sia essa naturale o per errore, senza però doverlo fare esplicitamente.

In Rust questo concetto andrà anche affiancato al concetto di proprietà. Ricordando che in C non esiste il concetto di proprietà di una variabile, un puntatore puo' essere usato in qualunque parte del mio programma. In Rust invece l'utilizzo di un puntatore andrà vincolato al concetto di proprietà e prestito, esclusivo o meno.

Prima di procedere a discutere gli aspetti pratici dei puntatori intelligenti offerti da Rust, notiamo che essi sono disponibili (più propriamente Rust implementa un super-set per questioni di cui parleremo più avanti) anche in C++ e hanno un funzionamento quasi del tutto equivalente.

Il primo puntatore di cui dobbiamo discutere è `RC`, che sono le iniziali di `Reference Counter`. Ne esiste anche una versione da utilizzare invece in ambienti concorrenti e che sono quindi thread-safe, `ARC`.

Il concetto è questo: tramite i meccanismi di copia (in Rust, detta `Clone` ricordiamo) tengo conto di quante volte io abbia condiviso la risorsa, ovvero quante parti del mio codice stiano attualmente utilizzando quel puntatore. All'atto del `drop` della risorsa, ovvero quando la sua variabile che contiene l'`RC` viene distrutta e le sue risorse eventualmente rilasciate, questo contatore verrà decrementato e al raggiungimento dello 0 la memoria puntata verrà liberata.

```
let x = RC::new(5);
let y = x.clone();
println!("{}", x.count()); // 2
drop(x);
println!("{}", y.count()); // 1
```

La variante `ARC` ha lo stesso identico funzionamento, ma risolve in modo differente l'opzione di decremento. Se infatti un programma in esecuzione su un singolo processore non avrò problemi ad aggiornare il contatore, questo potrebbe trovarsi in stati inconsistenti nel momento in cui più thread vi facciano accesso nello stesso momento. Per cui `ARC`, che sta per `Atomic Reference Counting`, utilizza solo istruzioni dette atomiche per la lettura e l'aggiornamento del contatore.

Questa caratterstica è raggiunta tramite l'uso di istruzioni, intese come istruzioni per processore, che eseguono la lettura, l'aggiornamento e la riscrittura del dato in un'unico colpo, ma richiedono molti cicli di clock per essere eseguite e forzano l'aggiornamento del dato in RAM e l'invalidazione della cache. Quindi

non è una buona idea utilizzare sempre e solo `ARC` in quanto riduce le nostre prestazioni: va utilizzato il puntatore adatto al nostro processo.

Il tipo `RC` va quindi ad essere funzionalmente indentico a quello che in C++ è il tipo `shared_ptr`.

In C++ un altro puntatore spesso utilizzato è lo `unique_ptr`, che invece assicura che del puntatore esista una ed una sola variabile che contiene la risorsa. In Rust questo meccansimo è superfluo per via del design del linguaggio, per via del concetto di possedimento del dato.

Ripetiamo infatti che in Rust a differenza del C un dato ha uno e un solo proprietario, e che tale proprietà è raggiunta mediante una differnziazione dell'operatore =: laddove in C `a=b` significherebbe una copia del dato `b` nella variabile `a` se non specificato diversamente, in Rust questo implica uno spostamento del dato di `b` in `a` se non specificato diversamente.

In C per ottenere la caratteristica di unicità si andrebbe quindi a sovrascrivere il funzionamento dell'operatore = per copiare la risorsa nel nuovo contenitore e cancellare la vecchia, cosa che Rust fa in automatico. Per ottenere quindi questa caratteristica in Rust, è sufficiente usare un normale puntatore (`Box<T>`) e non eseguirne mai la clone, ma solo assegnazioni.

```
let x = Box::new(5);
let y = x;
println!("{}", x); // Errore di compilazione qui: `x` è stato spostato!
```

Un'ultima categoria di puntatori è quella che in C sono i `weak_ptr`. Per capirli, è più veloce fare un esempio pratico: cosa accadrebbe se volessi implementare un buffer circolare mediante una lista ciclica?

Supponiamo il caso più banale con due nodi. `A->B` e quindi anche `B->A`. Notiamo una sottigliezza: essendo il buffer circolare, `B` ha a sua volta una referenza verso `A`, il che porta il conteggio delle referenze di `A` a 2.

Questo implica che in fase di liberazione del buffer, in fase di `drop()`, in seguito al `drop()` di `B` la sua referenza verso `A` viene distrutta ed il suo contatore non raggiungerà mai lo 0 e quindi non verranno mai liberate le sue risorse.

I puntatori deboli quindi offrono un modo di tenere traccia dei riferimenti senza che essi vadano a toccare i contatori di quante volte questo riferimento è stato condiviso, ma questo implica che non si puo' esser sicuri che il dato puntato sia valido al momento dell'utilizzo e quindi vada fatto un controllo preventivo.

Questo problema è risolto in Rust nuovamente mediante l'utilizzo di un `Option`. Funzionalmente, un `Weak` in Rust non è direttamente utilizzabile, ma ha prima bisogno di essere convertito in un `RC` vero e proprio.

La funzione `upgrade()` di `RC` infatti ritorna un `Option<RC>` che contiene, nel caso di dato valido, un vero e proprio `RC`, che ha incrementato i contatori e li decrementerà nuovamente quando verrà distrutto, oppure un None nel caso in cui questo dato non fosse valido.

Notiamo infine che, anche se disponibile un metodo `new` per la creazione diretta di un `Weak`, il modo più tipico per ottenerne uno è prima creare un `RC` e successivamente richiamare su esso la funzione `downgrade()` per ottenere il puntatore debole.

Commento personale: gli sviluppatori di Rust sono delle capocce geniali per aver prodotto sto linguaggio, eppure hanno ben pensato di non differenziare il nome dei `Weak` di tipo `RC` e quelli di tipo `ARC`. Quindi se dovete importare nello stesso file entrambi, dovete fare:

```
use std::rc::Weak as RCWeak;
use std::sync::Weak as ARCWeak;
```

Parliamo ora degli aspetti che Rust ha migliorato rispetto al C++ grazie al suo vantaggio di essere un linguaggi più moderno.

Una situazione tipica che potremmo incontrare è quella di aver bisogno di un dato immutabile ma che saltuariamente potremmo desiderare mutabile. Questo comportamento non è di base consentito, ma puo' essere raggiunto tramite i tipi `Cell` e `CellRef`.

Questo comportamento è necessario infatti perché i puntatori descritti prima permettono di condividere i dati con poca cura di come ciò accada, un concetto che mal si sposa con la filosifia di Rust. Per via di ciò, i dati puntati possono solo essere ottenuti come riferimento non mutabile, una restrizione non da poco.

Il tipo `Cell` è quindi un tipo di dato simile ad un puntatore ma che pur essendo immutabile, è accompagnato dal concetto di mutabilità interiore, ovvero il dato contenuto puo' eventualmente essere modificato.

Questo vale a dire che il concetto di proprietà di dato introdotto dal borrow checker in Rust non puo' esser fatto rispettare in fase di compilazione ma vada affrontato in fase di esecuzione. In Rust si fa ciò ad esempio spostando i valori all'interno e all'esterno di `Cell`.

Ci sono tre possibili alternative, a seconda del tipo `T` contenuto in `Cell<T>`: - T implementa `Copy` - Allora una copia di `T` viene restituita (esempio: `i32`) - T implementa `Default` - Allora una `take()` su `T` lascerà all'interno della `Cell` il valore `T::default()` - Per tutti gli altri casi: - `Cell::replace` permette di sostituire ad un dato `T` un altro dato di tipo `T`.

I tipi `Cell` e `RefCell` si differenziano in base al fatto che si vada a contenere un tipo di dato `T` oppure uno `&T`.

```
let cell = Cell::new(5); // notiamo: non mutabile
println!("{}", cell.get()); // 5
cell.set(7);
```

Un altro tipo molto utile per l'ottimizzazione di memoria sono i `Cow`. Essi infatti implementano il meccanismo di Copy (Clone) on Write, per cui permetto la condivisione di una zona di memoria fintanto che nessuno dei detentori della risorsa la modifica. Quando ciò accade, una copia esclusiva viene creata.

Infine, parliamo brevemente della dereferenziazione. In C avendo un puntatore possiamo accedere al dato puntato sia in lettura che scrittura tramite la derefernziazione, ottenuta con la notazione `*`. Posso quindi leggere il valore tramite `a=*b` ed assegnarlo come `*b=a`.

In Rust ovviamente questa operazione è ottenuta tramite due tratti, `Deref` e `DerefMut`, che permettono di eseguire queste due operazioni. Non c'è bisogno di chiamare le funzioni `deref()` o `deref_mut()`: l'operatore `*` funziona nella stessa identica maniera, ed anzi a volte il compilatore renderà superflua la sua chiamata a seconda dell'analisi statica del codice.

Tuttavia non tutti i puntatori supportano entrambe le operazioni. Di base un oggetto immutabile non esporrà `deref_mut()` ad esempio, ma alcuni tipi potrebbero non esperlo affatto. È il caso di `Cell` che essendo progettato per essere usato come estrazione-modifica-reinserimento, esso non espone nè `deref` nè `deref_mut`.

Va fatto notare che il compilatore di Rust ci viene incontro e ha funzionalità di coercizione della dereferenziazione. Questo significa che se ci dimenticassimo di applicarla ma sarebbe chiaro il nostro intento, esso espanderbbe il codice per ottenere il codice che avevamo intenzione di scrivere. In altre parole un codice del tipo:

```
let mut var : i32 = 0;
var = 10;
```

verrebbe automaticamente adattato dal compilatore come:

```
let mut var : i32 = 0;
*var = 10;
```