05-11.md

1. Lezione dell'11/05 –

Questa lezione non l'ho riscritta che mi pare inutile

Vi sarete accorti sicuramente con tanto di maledizione che il processo di gestione dei crate di Rust segue regole ben specifiche, ovvero quando scriviamo il nostro progetto e cerchiamo di mimare a distribuire il codice su più sorgenti, che Rust inizialmente sembra strano. Questo perchè il progetto per lui è l'equivalente di qualsiasi altra libreria esterna, ovvero è un crate che definisce una serie di cose che appunteremo così da sapere cosa c'è scritto nel TOML, e poi definisce tutte le librerie esterne e i sottomoduli che devono essere portati appresso.

COn un crate io posso avere due obiettivi, un'eseguibil oppure una libreria. La grossa differenza è abbastanza semplice, nel primo caso volgio qualcosa di eseguibile, nel secondo caso voglio qualcosa da includera altrove.

Supportando entrambe le cose, devo supportare sia librerie statiche che dinamiche e fare in modo che l'eseguibile deve richiederle correttamente. La differenza tra i due, ovvero statriche e dinamiche, è la dimensione, perchè se tutto è statico il codice finale include tutto il compilato più le librerie. Notiamo che non ho usato la parola funzione ma libreria. Nel C, se faccio un programma e lo compilo in modo statico, e uso la stdio, dentro l'eseguibile finale c'è tutta la stdio. Il vantaggio è che ho appresso tutto e quindi non devo chiedere nulla al SO, lo svantaggio è che se qualcosa che mi sto portando dietro non è 100% compatibile otterro qualche errore.

Di contro posso compilare con le librerie dinamiche, quindi con un eseguibile più piccolo, ma il sistema deve avere le librerie. Dall'altro lato, l'esecuzione è leggermente più lenta, perchè le librerie vengono caricate in memoria secondo un princio di lazy loading.

Queste sono differenze importanti per casi reali, ad esempio per sistemi embedded vorrò una compilazione statica in modo che la scheda non abbia bisogno di configurazione, lo stesso vale per i sistemi real time. In un general purpose invece vorrò che siano dinamiche, perchè non ha molto senso visto che un elefante che gira sul sistema che più o meno ha tutto.

Avremo notato che c'è la posisbilità di indicare nel toml la sezione `[lib]` che definische come viene eventualmente prodotta. Scrivendo una libreria in Python è possibile che io non possa utilizzarla in C. Questo perchè lo strumento con cui le libreria vengono messe in ondivsione in modo che siano trasversali ai linguaggi è quello del C, quindi rust ci chiede se la libreria sarà una libreria puramente rust, in modo che i simboli prodotti siano rust comptaibili e basta, oppure se deve essere una libreria compatibile col C, perchè in quel caso con un po' di maniplazione si aggiustino i simboli e vengano esportati in modo da esserre compatibili col C.

In termini pratici cambia il debug. Se decidessimo di fare una libreria C compatibile e la usassimo inizialmente solo in Rust troveremmo i simboli completamente diversi da come erano originariamente.

Per modulo rust intende sia fisicamente che virtualmente, che all'inzio da un po' di fastidio come dividere il codice, il nostro progetto. Questo è molto simile al namespace di C++ che è un modo per dare lo stesso nome alle funzioni ma sotto etichette diverse, quindi la libreria lista potrebbe avere due implmentazioni, le funzioni avaranno gli stessi nomi ma saranno disambiguati dal nome del modulo. Questo avviene in maniera esplicita utilizzando il nome `mod nome_modulo` oppure in modo fisico.

I moduli peremttono di avere sottomoduli che permettono di accedere alle funzioni private del modulo di livello superire. Gestendo la libreria standard anche i tratti sottostanno alle stesse regole. Quando scriviamo un metodo di un qualche tratto lui include usando la `use ...` una valanata di parole chiave che vanno importati perchè il linguaggio distingue da una versione di base, quindi librerie che vengono incluse automaticamente, da altre che invece vanno incluse in modo specifico, perchè il princio è che cioò che va

usato di frequente e che non puo' essere evitato non ha senso vada importato puntualmente, ma di contro qualsiasi compilazione si va ad eseguire è compilato staticamente e sempre presente nell'esguibile.

L'altra cosa è che come altri linguaggi, tutto il modulo puo' essere incluso in un'unica volta utilizzando `*`, perchè il principio è che la cartella dove è presente il main è la cartella del crate, quindi se la dentro creo un file e gli do un nome al cui al suo interno c'è delle un file `.rs` con delle funzioni, il nome del file automaticamente è il nome del modulo. È possibile invece creare sottocartelle con il nome desiderato del nostro modulo da usare a patto che il file in cui sia contenuto il codice si chiami `mod.rs`, dando un nome diveso staremmo creando un sottomodulo.

Chiaramente, c'è una parte `[dependencies]`, quella grazie all'IDE è di facile gestione, perchè permette di includere dei crate esterni in repository pubblcii in cui il crate, che tra l'altro possiamo a nostra volta pubblicare, è possibile importare questo create con l'unico vincolo che va specificata la versione, con tutti i simboli del caso per dire maggiore di, minore di, questa major realse, … Questo perchè anche il nostro eseguibile puo' essere sottoposto a versioning.

C'è una parte di `edition`. Questo perchè quella sorta di gruppo di librerie di base è definita dalla versione del linguaggio, che è in base all'anno, similmente al C++, che permette al compilatore di sapere quali librerie includere. Alcune funzioni presenti in una nuova versione potrebbero essere sperimental in una precedenza.

2. Testing

I test sono i tool che supportano il lingauggio di avere, con l'eseguibile, due blocchi di file extra. Il primo è quello che trattiamo, il secondo non lo vedremo che sono i benchmark, che sono un po' meno importanti dei test, in quanto sono loro che ci permettono di verificare la correttezza del nostro codice.

Per il C il testing è complesso, esistono molti standard per implementare i test spesso con strane limitazioni, come il MISRA C che non permette di usare più di 5 parametri.

Il principio è mettere insieme delle funzionalità da eseguire prima della compilazione globale, perchè se prendo il mio programma e ha un'unico main che legge qualche file ma si basa sul caricamento di dati da file da una struttura dati, è più facile garantire il funzinamento se prima ho testato la struttura dati che non derivare da qualche test manuale le funzionalità, poichè il programma principale potrebbe non trovare tutti i difetti.

I test sono tipicamente pensati per dividere il problema in funzioni o funzionalità e per ognuna di essa immagino quali test fare. L'utiltià è duplice, ovvero in fase di produzione e in fase di mantenimento. Vorrei infatti esser certo di non rompere parti di codice funzionante nel momento in cui vado a modificare il mio eseguibile per aggiungere funzionalità o correggere altre parti di codice.

Un'altro cappello sul test è quello di integrazione. Qui andiamo a toccare il fatto che un programma non sia composto da un singolo componente, ma da più parti messe assieme. Quello che faccio con i test di integrazione non è testare la singola funzionalità ma come le parti del mio sistema comunicano.

I test di unità in rust hanno più opzioni. Posso prednere il mio modulo e riempirlo con la configurazione di `#[cfg(test)]`, quindi i test sono assieme al codice sorgente, che va bene finchè il codice è piccolo. Ma quando mi faccio una libreria con tante funzioni e tanti test questo non scala.

Invece possiamo creare una cartella di test, in cui inserire i miei test. RIcordiamo che i test di unità servono a testare il modulo, quindi un errore comune è che questo modulo non stia venendo importato correttamente.

I test di unità si basano sul richiamare funzioni o funzionalitò dalla mia libereria e tramite una serie di `assert` far fallire i test al primo fallimento. Queste assert impongono il vero funzinamento del test, quindi cerchiamo di non usare solo la `assert!` base ma anche le varianti `assert_eq!` o `assert_ne!` anzichè fare `assert!(val1 == val2)`.

Per eseguire i test è sufficiente richiamare da riga di comando `cargo test`. Notate che ad esempio su exercism tutto è organizzato tramite test, che segue lo schema classico di sviluppo quindi la maggior parte dei test è pronta ma saltata tramite le macro di `#ignore`. Spesso e volentieri l'approccio da seguire è di tentare di passare prima i test più semplici e poi quelli più complicati dopo.