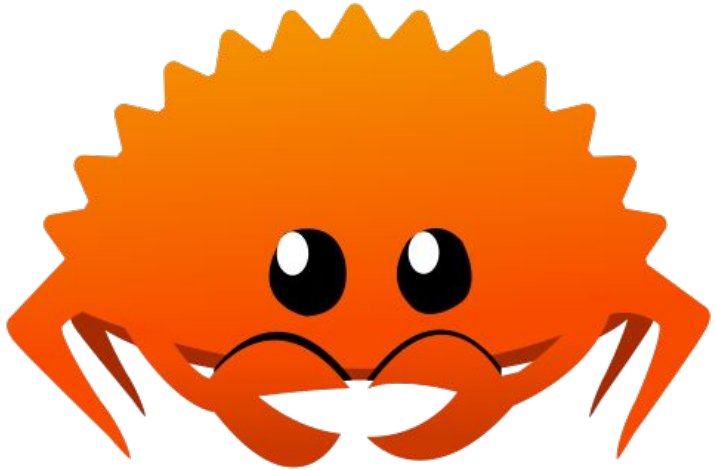
UNIVERSITÀ
DEGLI STUDI DI
PADOVA

MTSS 2021-2022

ALESSIO
FERRARINI

ELIA
PASQUALI

ANALISI DEL PROGETTO RUST



COS'È RUST

Rust è un linguaggio di programmazione compilato che ha l'obiettivo di essere performante, sicuro e memory safe senza utilizzare un garbage collector.

È un progetto di software libero ed open-source, distribuito con doppia licenza MIT e Apache 2.0 entrambe permissive.

Il progetto è disponibile su GitHub all'indirizzo https://github.com/rust-lang/rust, il repository GIT oltre al compilatore contiene la libreria standard del linguaggio, la sua documentazione e i vari file di configurazione per gli strumenti di CI e CD.

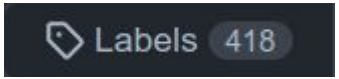
ISSUE TRACKING SYSTEM

Il progetto fa ampio uso dell' issue tracking system integrato all'interno di GitHub (https://github.com/rust-lang/rust/issues) data la facile integrazione con le GitHub-Action e i Bot della sezione discussion, questo è dovuto al grande focus messo sull' automazione del core team di maintainer che in progetto open source di queste dimensioni risulta essere fondamentale.

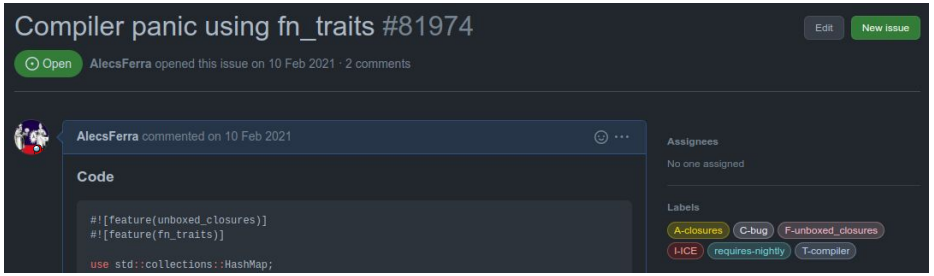
Al momento sono aperte più di 5000 issue e 3 milestone.

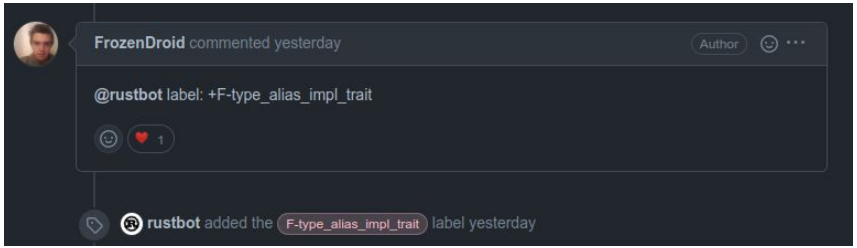
IL SISTEMA DI LABELING

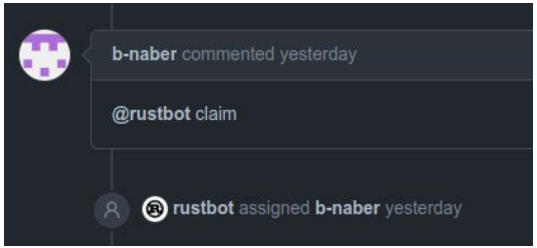
A per indicare l'area di riferimento di tale issue
B per indicare dei bug identificati come bloccanti
beta per indicare cambiamenti da riportare nei branch beta
C per indicare la categoria della issue
E serve ad indicare il livello di esperienza necessaria al fix di tale issue
final-comment-period indica dei bug che attendono di essere commentati tramite il sistema di RFC
I specifica il livello di importanza della issue, può essere unita ad altre per indicare se verrà discussa nel prossimo team meeting
metabug racchiudono liste di bug raccolte in altre categorie
O indicano a quale sistema operativo fa riferimento la issue
P per indicare la priorità del bug. Sono assegnate da chi può capire tale issue e rimpiazzare le label I
proposed-final-comment-period definisce il periodo precedente all'entrata nel final-comment-period
regression traccia le regressioni dalla versione stabile ai canali di release
relnotes servono ad indicare quali cambiamenti andrebbero documentati nelle note della successiva release
S per tracciare lo stato delle pull request
T per indicare a quali team la issue è assegnata

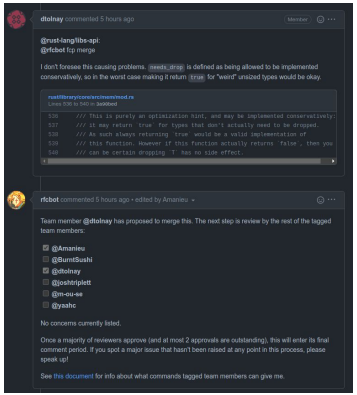


Inoltre sono presenti delle label extra utilizzate in modo temporaneo per feature che vanno a cambiare radicalmente la struttura del progetto.









APRIRE UNA ISSUE

All'interno del repository sono presenti dei template per aprire le issue in modo che un utente possa segnalare un bug o richiedere una feature in modo corretto anche senza conoscere a pieno le norme interne al progetto e comunque sfruttando a pieno le varie automazioni con i bot.

Inoltre quando determinate label vengono aggiunte triage bot si occupa di pingare i team interessati su Zulip.

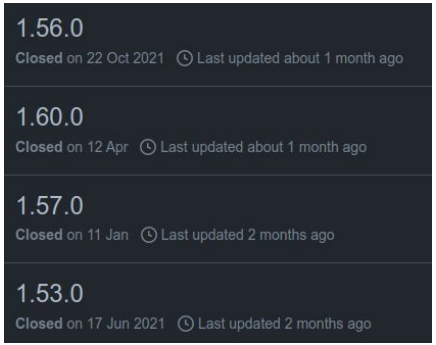
MILESTONE

Le milestone in questo progetto vengono utilizzate per raccogliere le task e collegarle ad una determinata versione del prodotto.

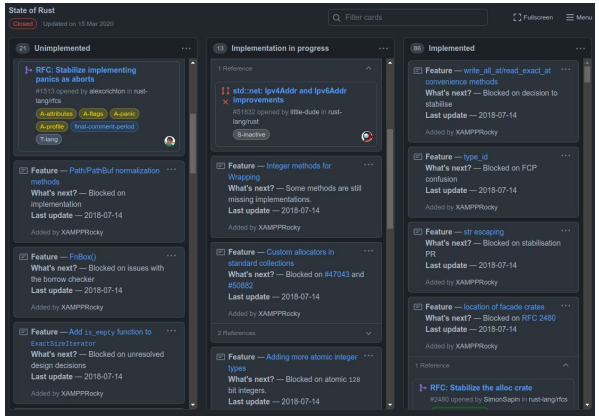
Si nota che le varie issue vengono collegate alle rispettive milestone, esse tramite le label descrivono il loro stato corrente.

Le milestone mancano di date di scadenza, potrebbero essere aggiunte dato che il rilascio segue uno schema temporale ben definito.

In ogni istante di tempo sono aperte 3 milestone che corrispondono al rilascio stable, beta e nightly.



PROJECT BOARD



Il progetto attualmente non fa più uso di una project board.

Nelle vecchie board chiuse si nota in particolare "*State of Rust*", che è stata chiusa mentre alcuni lavori non erano ancora stati completati.

La decisione di abbandono è dovuta al fatto che lo strumento integrato in GitHub non è abbastanza flessibile per lo sviluppo di uno strumento così complesso e a cui collaborano così tante persone invece si è trovato il sistema di milestone, label e bot al sistema di rilascio e di lavoro di Rust.

MODELLO DI RILASCIO

Il rilascio delle nuove versioni del linguaggio segue il *modello a treno*.

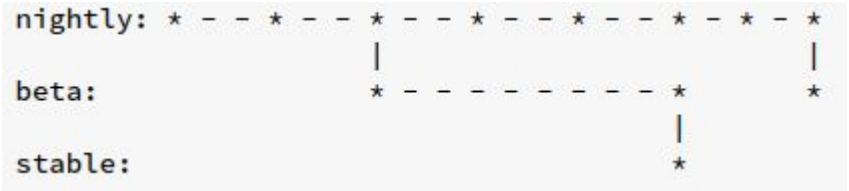
Tutto lo sviluppo avviene sul master branch del repository e sono presenti tre canali di rilascio:

- *Nightly* dove ad ogni notte viene rilasciato l'attuale stato del master branch.
- *Beta* dove ogni sei settimane viene staccata una versione beta del linguaggio.
  Se a questo punto del rilascio vengono trovati dei problemi, alla correzione saranno reinseriti nel master branch in modo da averli sia nelle *nightly build* che nella futura *stable release*.
- *Stable* dopo sei settimane dal rilascio della beta una versione stabile viene rilasciata.

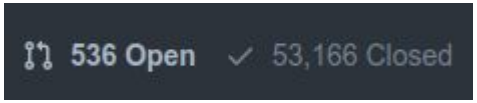
Tutte le versioni stabili sono disponibili sulla pagina delle release di Github e sul sito ufficiale del progetto (dove può essere installato con il tool rustup), insieme al changelog e alle note di compatibilità, il tutto è spiegato all'interno della documentazione.

Per la numerazione delle versioni è utilizzato lo standard semver.

Viene chiamato *modello a treno* perché ogni sei settimane una nuova versione "lascia la stazione", ma deve continuare il viaggio attraverso il canale beta prima di diventare una versione stabile



CONTRIBUZIONI



Il progetto propone vari modi per contribuire allo sviluppo, ad esempio

- Feature Request
- Bug Report
- Pull Request
- Scrittura della documentazione
- Issue triage, cioè il controllo e la categorizzazione delle issue presenti. Definizione del tipo e della priorità e chiusura di quelle risolte e lasciate aperte
- Risposte alle domande su siti esterni, nel server Discord del progetto e la partecipazione attiva a tutto il processo di RFC.
- Pubblicazione di progetti e librerie su crates.io

PULL REQUEST

Il progetto utilizza il modello di *fork and pull* per la gestione delle PR, come descritto nella documentazione di Github.

Questo approccio permette al contributore di creare un suo fork personale della repo, che verrà poi inviato tramite una pull request ai gestori del progetto, che decideranno se integrarlo o meno.

Tutti coloro con i permessi di push sull'upstream possono aggiungere cambiamenti alla pull request.

Questo modello riduce le frizioni tra i nuovi contributori e permette alle persone di lavorare indipendentemente.

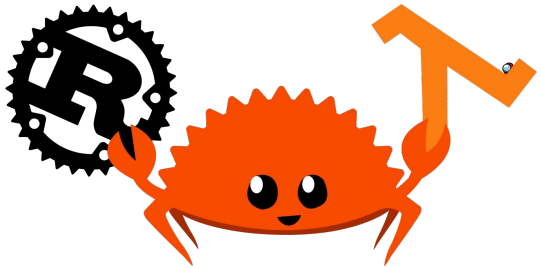
Guida alle PR del linguaggio: https://rustc-dev-guide.rust-lang.org/contributing.html

VERSION CONTROL SYSTEM

Il versionamento del codice del progetto Rust viene gestito tramite Git, un sistema di versionamento distribuito, sfruttando la piattaforma Github.

Lo sviluppo avviene tramite il Forking Workflow. Si differenzia dagli altri Git Flow perché invece di utilizzare un singolo repository centralizzato su cui lavorare, le modifiche avvengono nei *fork* privati, che verranno integrate successivamente tramite merge a seguito di una pull request.

Una modifica particolare è la No-Merge Policy, anche detto *rebase workflow*, questo è stato deciso per avere una history dei commit più lineare e permettere di effettuare bisect in modo più agevole, cosa molto comune all'interno di un software così delicato come un compilatore.



Tipico ciclo di vita di una contribuzione

Quando una nuova feature deve passare da una RFC sulla repository, questa viene implementata e dopo che è sufficientemente matura può essere stabilizzata e portata all'interno del progetto.

Per quanto riguarda invece tutto il resto e l'implementazione descritta prima:

1. Si reclama una issue con il comando `@rustbot claim`
2. Si sviluppa una PR come descritto precedentemente e si chiede una review tramite `@rustbot r? <utente>` / `<gruppo del compiler team>`e ne verrà assegnato uno casualmente
3. Viene eseguito il processo di CI, molto lungo e complesso a causa delle molteplici architetture e SO da supportare
4. Infine tramite il reviewer `@bors r+` per segnalare al bot di integrazione `bors` di iniziare il merge su master della PR. Se il contenuto è della PR è abbastanza piccolo viene usato `r+ rollup` che unisce più pull request che modificano componenti separati del compilatore per eseguire un unico merge una volta sola.

BUILD AUTOMATION

Dal README del progetto viene spiegato come eseguire la build del sorgente sui vari sistemi operativi

Viene fatto uso di uno script Python, *rustbuild*, che si occupa della build del compilatore che gestisce l'avvio di tutto il processo.

Vengono specificate anche le dipendenze necessarie, dato che lo script non si occupa della loro installazione automaticamente.

È possibile ottenere la documentazione sempre tramite lo stesso strumento.

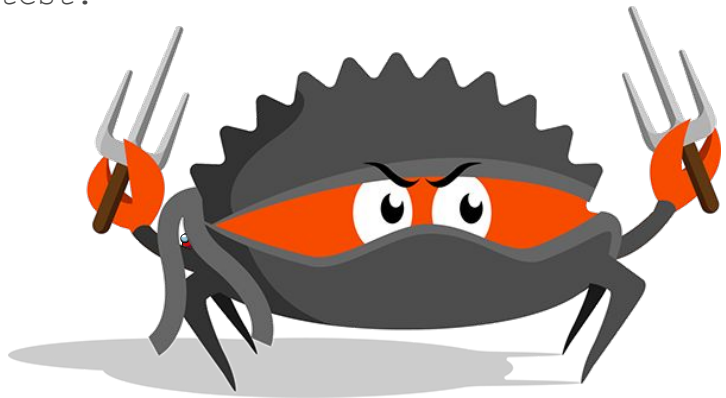
Al cuore di tutta la build c'è lo strumento Cargo, utilizzato tipicamente all'interno di tutti i progetti Rust, siccome il compilatore è suddiviso in più *crates* per ognuna è presente una sua configurazione di cargo specifica.

TEST

I test di unità sono condivisi nel VCS nella la cartella `/src/test`. È possibile eseguirli richiamando lo script di build con il flag test, oppure aggiungendo la cartella con i test specifici.

La test suite di Rust va a coprire `rustc`, la standard library, il package manager `cargo`, `rustdoc` e la documentazione. Altri test vanno a coprire poi casi specifici e bug rilevanti per Rust.

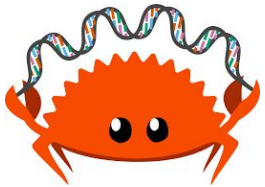
Tutti i test sono basati su `libtest`, un *crate* standard del linguaggio e coordinati dallo strumento `compiletest`.



Link ad articoli sui test in Rust

- Testing the compiler
- How Rust is tested

TIPOLOGIE DI TEST - COMPILETEST



I `compiletest` sono i principali test di Rust. Questo strumento permette di gestire un grande numero di test, con un'efficiente esecuzione parallela di questi e permette all'autore di configurare il comportamento atteso sia dai test individuali che da quelli di gruppo.

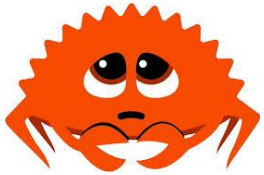
I test sono organizzati come semplici sorgenti Rust con delle annotazioni nei commenti che verranno utilizzate direttamente da `compiletest` per eseguirli.

Alcune test suite disponibili sono:

- `ui` per controllare stdout/stderr delle compilazioni e i il funzionamento degli eseguibili
- `pretty` per il controllo sul pretty printing dei risultati
- `incremental` per la compilazione incrementale
- `codegen` e `codegen-units` per la generazione di codice
- vari test per `Rustdoc`

Sono presenti anche dei test per `Valgrind`. Questi non sono più utilizzati nel processo di CI e come specificato nella documentazione potrebbero essere rimossi in futuro.

TIPOLOGIE DI TEST - Package test



La libreria standard e molti pacchetti del compilatore includono tipici unit test di Rust (`#[test]`), test di integrazione e i test della documentazione.

Sempre tramite il comando `x.py test <modulo>`. Questo eseguirà `cargo test` su quel modulo.

Tutti i test sono inseriti in file separati. Questo approccio assicura che quando uno di questi viene cambiato non è necessario eseguire nuovamente la build dell'intero crate.

Command	Description
`./x.py test library/std`	`Runs tests on std`
`./x.py test library/core`	`Runs tests on core`
`./x.py test compiler/rustc_data_structures`	`Runs tests on rustc_data_structures`

TIPOLOGIE DI TEST - Stile, formattazione e altri tool

Tidy

Tidy è uno strumento usato per validare lo stile del codice e le convenzioni di formattazione, ad esempio rifiutando linee di codice che superano una determinata lunghezza.

Formatting

Rustfmt è integrato nel sistema di build per obbligare ad utilizzare uno stile uniforme all'interno del compilatore.

I controlli di formattazione sono richiamati ed eseguiti automaticamente dallo strumento Tidy

Tool testing

I pacchetti inclusi nel progetto Rust vengono giustamente testati.

Tutti i prodotti presenti nel repository come cargo, clippy, rustfmt, rls, miri, bootstrap, etc hanno le loro suite di test.

TIPOLOGIE DI TEST - Test su infrastrutture esterne

Crater

Crater è uno strumento che esegue i test su tutti i crate presenti su crates.io

Viene eseguito quando si sta cercando di introdurre una funzionalità che potrebbe portare dei cambiamenti critici per molti pacchetti esistenti.

Test di performance tramite `rustc-perf`

Dato che uno dei focus principali del progetto è mantenere un compilatore efficiente si cerca di evitare regressioni di performance.

Tramite rustc-perf si può ottenere una completa reportistica. Le esecuzioni di questo strumento avvengono su un'infrastruttura separata, dove si tracciano, al momento, solo le build su `x86_64-unknown-linux-gnu`.

I risultati sono una comparazione tra due versioni del compilatore, identificate dall'hash dei loro commit

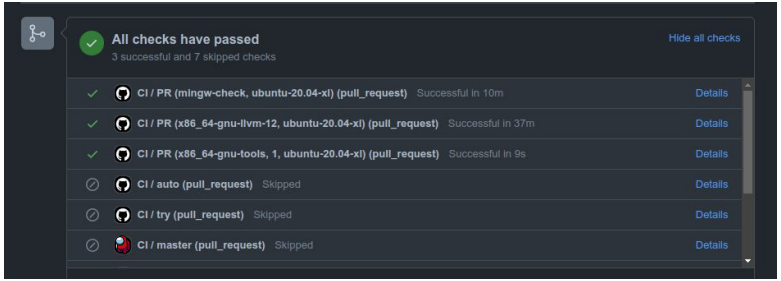
CONTINUOUS INTEGRATION

Al momento la CI del progetto è duplicata, per essere eseguita sia da Azure Pipeline che da Github Actions.
Questa è una situazione temporanea in quanto si passerà solamente a GH actions.

Entrambe sono generate automaticamente e utilizzano uno script all'interno del repository che, tramite il tool `expand-yaml-anchors`, rende tutto il più indipendente possibile da una singola piattaforma.

I vari passaggi svolti dalla Action in Github sono eseguiti per ogni combinazione di sistema operativo e architettura supportati, sfruttando il meccanismo della matrix per parallelizzare il lavoro. Viene mantenuta una cache per alleggerire le esecuzioni.

1. Instanzia il container configurato per rispecchiare il sistema in esame
2. Esegue la build del progetto tramite lo script
3. Esegue tutti i test disponibili descritti in precedenza
4. Al termine, avvisa Bors dello stato della Action



How the Rust CI works:
https://forge.rust-lang.org/infra/docs/rustc-ci.html

ARTIFACT REPOSITORY

I rilasci degli artefatti di tutto l'ecosistema Rust avviene su crates.io. Il package manager del linguaggio, che si occupa di gestire questi *crate* è Cargo.

Cargo si occupa di scaricare pacchetti, dipendenze, compilare e distribuire, anche tramite upload sul registro dei pacchetti della comunità. Anche questo strumento è open-source e disponibile su Github.

Come per Maven e simili, è possibile sfruttare Cargo per gestire i build lifecycle dei propri progetti. Tutte le configurazioni del progetto sono definite in un file `Cargo.toml`

Struttura di un Cargo.toml

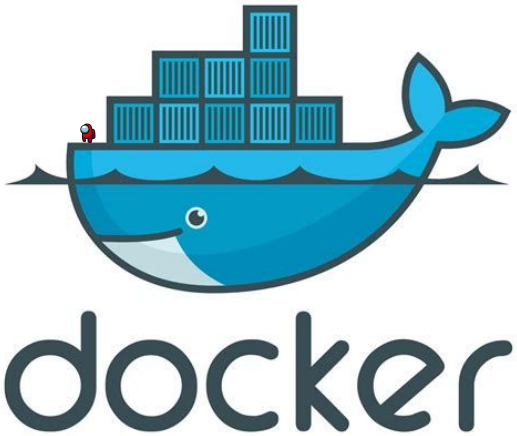
```
[package]
name = "hello_world"
version = "0.1.0"
edition = "2021"

[dependencies]
```

CONFIGURATION MANAGEMENT

Lo strumento utilizzato per fornire un ambiente preconfigurato con tutte le dipendenze necessarie alla toolchain di bootstrap sono dei container Docker.

All'interno del repository sono presenti le cartelle contenenti i Dockerfile per la creazione degli ambienti e gli script bash per avviare la compilazione.



CONTINUOUS DELIVERY

Ogni notte viene rilasciata una versione utilizzabile del linguaggio e dei suoi strumenti nel canale Nightly.

Il prodotto é rilasciato a partire dall'ultimo commit su master. Dato il processo di build e Continuous Integration precedente a quel commit si ha la sicurezza che questo è utilizzabile .

Questo può essere quindi considerato un processo di Continuous Delivery