Programmieren mit Rust

Thema 2: Cargo

Dirk Müller und Robert Baumgartl

22. März 2024

Cargo in a Nutshell

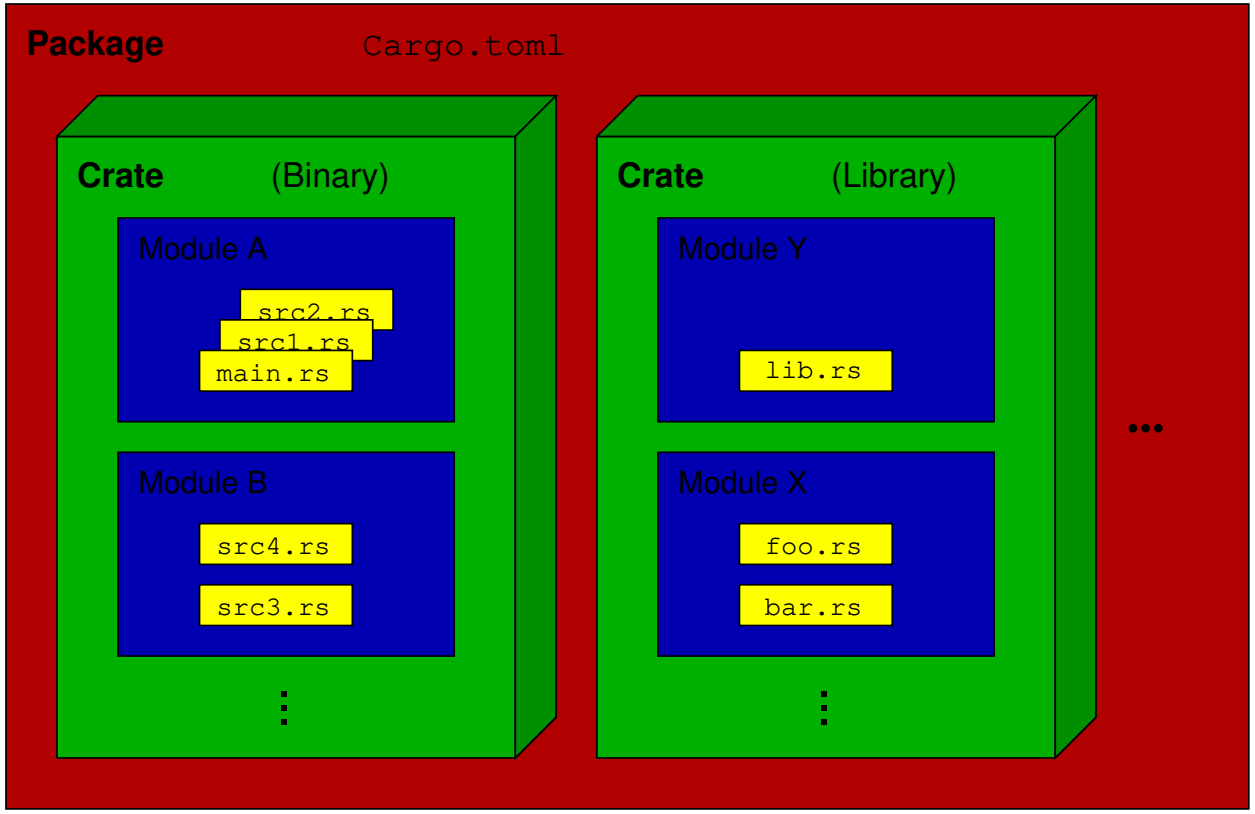
Build Tool + Paket-Manager von Rust; zum

- Laden von Paketabhängigkeiten,
- Kompilieren,
- Erstellen von Paketen,
- Hochladen von Paketen zu `crates.io`
- ...

Begrifflichkeiten

- unterste Ebene der Hierarchie: Funktion
- Quelldatei: Namenssuffix `.rs`; enthält i. d. R. mehrere Funktionen
- Modul
- Crate („Kiste“) mit Quellcode-Wurzeldatei und ggf. weiteren Modulen
  - `main.rs` → Binary (ausführbare Datei)
  - `lib.rs` → Library (Bibliothek)
- Paket besteht aus einem oder mehreren Crates
  - `Cargo.toml` – beschreibt Abhängigkeiten
- ein oder mehrere Pakete können in einem so genannten *Workspace* organisiert werden

Struktur eines Rust-Projektes



Vergleich mit C

- Funktionen (einen oder keinen Resultatwert, bel. Anzahl Argumente)
- Hauptfunktion/Einsprungpunkt: `main()`
- Quelltextdateien (`.c`)
- Headerdateien (`.h`)
- Bibliotheken (Binaries und Headerdaten – `lib/include`
- keine integrierten Werkzeuge zur Projektverwaltung ($\rightsquigarrow$ `make`)

Anlegen eines neuen Projekts

```
robge@sorpen:~/src/rust$ cargo new hello_cargo
     Created binary (application) `hello_cargo` package
robge@sorpen:~/src/rust$ tree -a hello_cargo
hello_cargo
├── Cargo.toml
├── .git
│   ├── config
│   ├── description
│   ├── HEAD
│   ├── hooks
│   │   ├── applypatch-msg.sample
│   │   ├── commit-msg.sample
│   │   ├── fsmonitor-watchman.sample
│   │   ├── post-update.sample
│   │   ├── pre-applypatch.sample
│   │   ├── pre-commit.sample
│   │   ├── pre-merge-commit.sample
│   │   ├── prepare-commit-msg.sample
│   │   ├── pre-push.sample
│   │   ├── pre-rebase.sample
│   │   ├── pre-receive.sample
│   │   ├── push-to-checkout.sample
│   │   └── update.sample
│   ├── info
│   │   └── exclude
│   ├── objects
│   │   ├── info
│   │   └── pack
│   └── refs
│       ├── heads
│       └── tags
├── .gitignore
└── src
    └── main.rs

11 directories, 20 files _
```

- legt ein neues Projekt (*Package*) mit dem angegebenen Namen sowie eine Verzeichnishierarchie im aktuellen Vz. an
- Name des Packages: `hello_cargo`
- Konfigurationsdatei: `Cargo.toml`
- initialisiert git-Repository (`.git/`, `.gitignore`)
- Verzeichnis für Quelldateien: `src/`
- Quelldatei (Stub) `src/main.rs`

Konventionen

- `src/` für alle Quelltextdateien
- `target/` während des Bauens generierte Dateien (z. B. das Binary)
- in die Projektwurzel gehören:
  - Konfigurationsdateien (`Cargo.toml`, `Cargo.lock`)
  - ggf. `README`
  - Informationen zur Lizenzierung
  - alles, was nicht codespezifisch ist

Nutzung: Übersetzen und Ausführen eines Projektes

```
robge@sorpen:~/src/rust$ cd hello_cargo/
robge@sorpen:~/src/rust/hello_cargo$ cargo run
   Compiling hello_cargo v0.1.0 (/home/robge/src/rust/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 3.97s
     Running `target/debug/hello_cargo`
Hello, world!
robge@sorpen:~/src/rust/hello_cargo$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/hello_cargo`
Hello, world!
```

- zweiter Aufruf ist schneller, da nicht übersetzt und gelinkt werden muss
- es werden (ähnlich wie `make`) nur diejenigen Quelldateien übersetzt, deren Modifikationszeitstempel größer ist als der des Zieles

Situation nach erfolgreicher Übersetzung

```
robge@sorpen:~/src/rust/hello_cargo$ tree -a
.
├── Cargo.lock
├── Cargo.toml
├── .git
│   ├── config
│   ├── description
│   ├── HEAD
│   ├── hooks
│   │   ├── applypatch-msg.sample
│   │   ├── commit-msg.sample
│   │   ├── fsmonitor-watchman.sample
│   │   ├── post-update.sample
│   │   ├── pre-applypatch.sample
│   │   ├── pre-commit.sample
│   │   ├── pre-merge-commit.sample
│   │   ├── prepare-commit-msg.sample
│   │   ├── pre-push.sample
│   │   ├── pre-rebase.sample
│   │   ├── pre-receive.sample
│   │   ├── push-to-checkout.sample
│   │   └── update.sample
│   ├── info
│   │   └── exclude
│   ├── objects
│   │   ├── info
│   │   └── pack
│   └── refs
│       ├── heads
│       └── tags
├── .gitignore
├── src
│   └── main.rs
└── target
    ├── CACHEDIR.TAG
    ├── debug
    │   ├── build
    │   ├── .cargo-lock
    │   ├── deps
    │   │   ├── hello_cargo-998b8934749f5499
    │   │   └── hello_cargo-998b8934749f5499.d
    │   ├── examples
    │   ├── .fingerprint
    │   │   └── hello_cargo-998b8934749f5499
    │   │       ├── bin-hello_cargo
    │   │       ├── bin-hello_cargo.json
    │   │       ├── dep-bin-hello_cargo
    │   │       └── invoked.timestamp
    │   ├── hello_cargo
    │   ├── hello_cargo.d
    │   └── incremental
    │       └── hello_cargo-3etgsa1g3etfy
    │           ├── s-gucztwgm0q-1dkgdw-33swp4a5c3dfsd3zx5pkwj4pa
    │           │   ├── 11ayjtlpiuo6r84v.o
    │           │   ├── 1i4ec46rvtjxbzuq.o
    │           │   ├── 42lsvxx381wqqu04.o
    │           │   ├── dep-graph.bin
    │           │   ├── lvpd99zpnztexih.o
    │           │   ├── query-cache.bin
    │           │   ├── s37erzh6s4ltu8q.o
    │           │   ├── tm4nx1qaj7pxodj.o
    │           │   └── work-products.bin
    │           └── s-gucztwgm0q-1dkgdw.lock
    └── .rustc_info.json
```

- 21 Verzeichnisse, 42 Dateien, ≈ 4 MiB
- Vz. `target/` neu hinzugekommen
- + `Cargo.lock`
- Standardname der neu entstandenen ausführbaren Datei: `target/debug/<projektname>`
- über Pfad natürlich auch Ausführung des Binaries möglich

Erzeugung einer Release-Version

```
robge@sorpen:~/src/rust/hello_cargo$ cargo build --release
   Compiling hello_cargo v0.1.0 (/home/robge/src/rust/hello_cargo)
    Finished release [optimized] target(s) in 0.77s
robge@sorpen:~/src/rust/hello_cargo$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    ├── CACHEDIR.TAG
    ├── debug
    │   ├── build
    │   ├── deps
    │   │   ├── hello_cargo-998b8934749f5499
    │   │   └── hello_cargo-998b8934749f5499.d
    │   ├── examples
    │   ├── hello_cargo
    │   ├── hello_cargo.d
    │   └── incremental
    │       └── hello_cargo-3etgsa1g3etfy
    │           ├── s-guij18rwjy-zu1yau-33swp4a5c3dfsd3zx5pkwj4pa
    │           │   ├── 11ayjtlpiuo6r84v.o
    │           │   ├── 1i4ec46rvtjxbzuq.o
    │           │   ├── 42lsvxx381wqqu04.o
    │           │   ├── dep-graph.bin
    │           │   ├── lvpd99zpnztexih.o
    │           │   ├── query-cache.bin
    │           │   ├── s37erzh6s4ltu8q.o
    │           │   ├── tm4nx1qaj7pxodj.o
    │           │   └── work-products.bin
    │           └── s-guij18rwjy-zu1yau.lock
    └── release
        ├── build
        ├── deps
        │   ├── hello_cargo-ac6d8d77e18d4d6b
        │   └── hello_cargo-ac6d8d77e18d4d6b.d
        ├── examples
        ├── hello_cargo
        ├── hello_cargo.d
        └── incremental

15 directories, 22 files
```

- 28 Verzeichnisse, 51 Dateien, $\approx$ 7.5 MiB
- erzeugt weiteres Binary in `target/release`
- längerer Buildprozess, da Optimierung
- dafür schnellere Ausführung
- enthält immer noch Debug-Information ($\rightsquigarrow$ Kdo. `strip`)
- Größe der Binarys differiert kaum
- BEWARE: Overflows werden nicht mehr erkannt!

`Cargo.toml` zur Konfiguration des Projektes

- als *Manifest* bezeichnet
- wird automatisch beim `cargo new` angelegt, kann und soll manuell angepasst werden
- enthält Metadaten, die für die Übersetzung des Projektes erforderlich sind
- Key-Value-Paare, leicht zu parsen

= Tom's Obvious Minimal Language

- strukturiert in Sektionen
  - `[package]`: Name des Projektes, aktuelle Version, Rust-Edition (bislang: {2015, 2018, 2021}), Programmautor
  - `[dependencies]`: für das Projekt benötigte andere *Crates* (Bibliotheken)

`Cargo.toml`

Beispiel

```
[package]
name = "game-of-life"
version = "0.1.0"
edition = "2021"

[dependencies]
ggez = "0.7"
rand = "0.8"
```

-
- 2 externe Crates benötigt:
  - `ggez` („Good Games Easily“, https://ggez.rs/)
  - `rand` (Zufallszahlengenerator)

`Cargo.lock`

- während des Build-Prozesses aus `Cargo.toml` generiert
- enthält die exakten Versionsinformationen für das Bauen des Paketes ($\rightsquigarrow$ Build mit gleicher `Cargo.lock` führt zu identischem Binary) – Ziel: *Deterministic Builds*
- darf nicht manuell bearbeitet werden
- ist ein Snapshot aller konkreten Abhängigkeiten eines erfolgreichen Build-Prozesses
- `cargo update` aktualisiert `Cargo.lock`

Beispiel für `Cargo.lock`

```
# This file is automatically @generated by Cargo.
# It is not intended for manual editing.
version = 3

[[package]]
name = "ab_glyph"
version = "0.2.23"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "80179d7dd5d7e8c285d67c4a1e652972a92de7475beddfb92028c76463b13225"
dependencies = [
 "ab_glyph_rasterizer",
 "owned_ttf_parser 0.20.0",
]

[[package]]
name = "ab_glyph_rasterizer"
version = "0.1.5"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "a13739d7177fbd22bb0ed28badfff9f372f8bef46c863db4e1c6248f6b223b6e"

[[package]]
name = "adler"
version = "1.0.2"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "f26201604c87b1e01bd3d98f8d5d9a8fcbb815e8cedb41ffccbeb4bf593a35fe"

[[package]]
name = "alsa"
version = "0.6.0"
source = "registry+https://github.com/rust-lang/crates.io-index"
checksum = "5915f52fe2cf65e83924d037b6c5290b7cee097c6b5c8700746e6168a343fd6b"
dependencies = [
 "alsa-sys",
 "bitflags",
 "libc",
 "nix 0.23.2",
]
```

Versionsnumerierung

Frage: Wie sollten Softwareprojekte und deren Teile versioniert werden, wenn viele Abhängigkeiten bestehen?

- spezifiziert man die Abhängigkeiten sehr strikt und kleinschrittig, dann zieht die Aktualisierung eines Paketes P die Aktualisierung aller von P abhängigen Pakete nach sich (*Version Lock*)
- spezifiziert man die Abhängigkeiten zu großzügig, dann gibt ein Paket P nur noch vor, kompatibel zu den von ihm abhängigen Paketen zu sein, ist es aber in Wahrheit nicht mehr (*Version Promiscuity*)

$\rightarrow$ „Dependency Hell“

Vorschlag zur Standardisierung: **Semantic Versioning**

Semantic Versioning

Def. Versionsnummern werden in der Form

`MAJOR.MINOR.PATCH`

angegeben.

Jedes Modul besitzt eine API (Appplication Programmer's Interface; die Gesamtheit der durch andere Module nutzbaren Funktionen).
Regeln:

1. `MAJOR` wird inkrementiert, wenn die API inkompatibel verändert wird.
2. `MINOR` wird inkrementiert, wenn Funktionalität addiert wird, so dass die API kompatibel bleibt.
3. `PATCH` wird verändert, wenn kleine (rückwärtskompatible) Änderungen (z. B. Bugfixes) erfolgen.

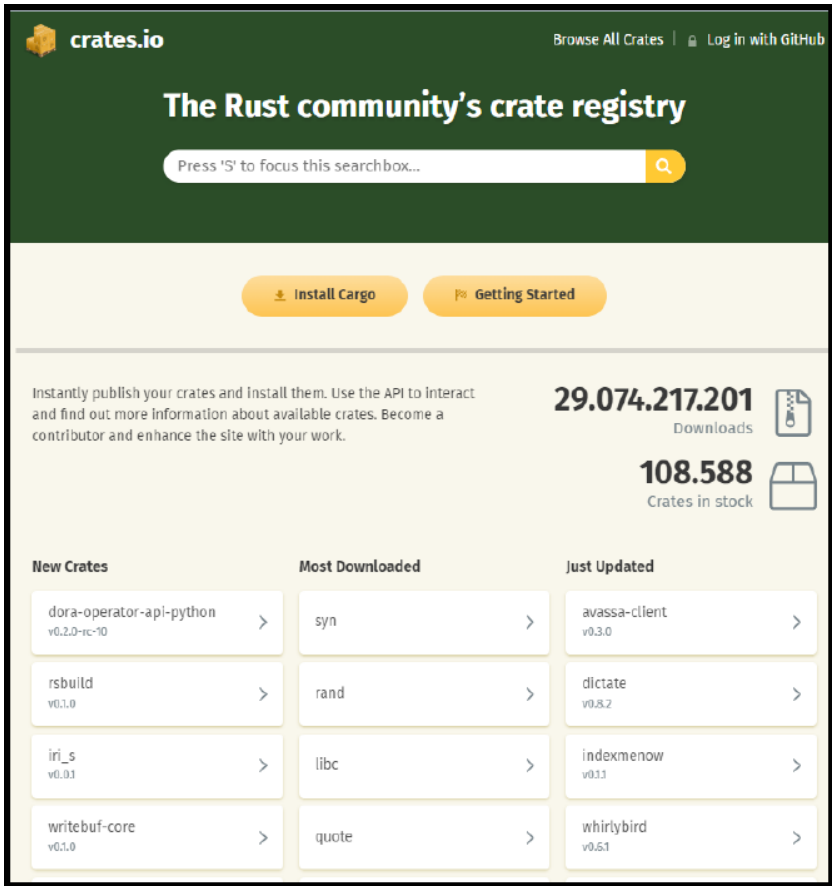
Jedes Modul startet mit Version `0.1.0`.

Speicherbedarf

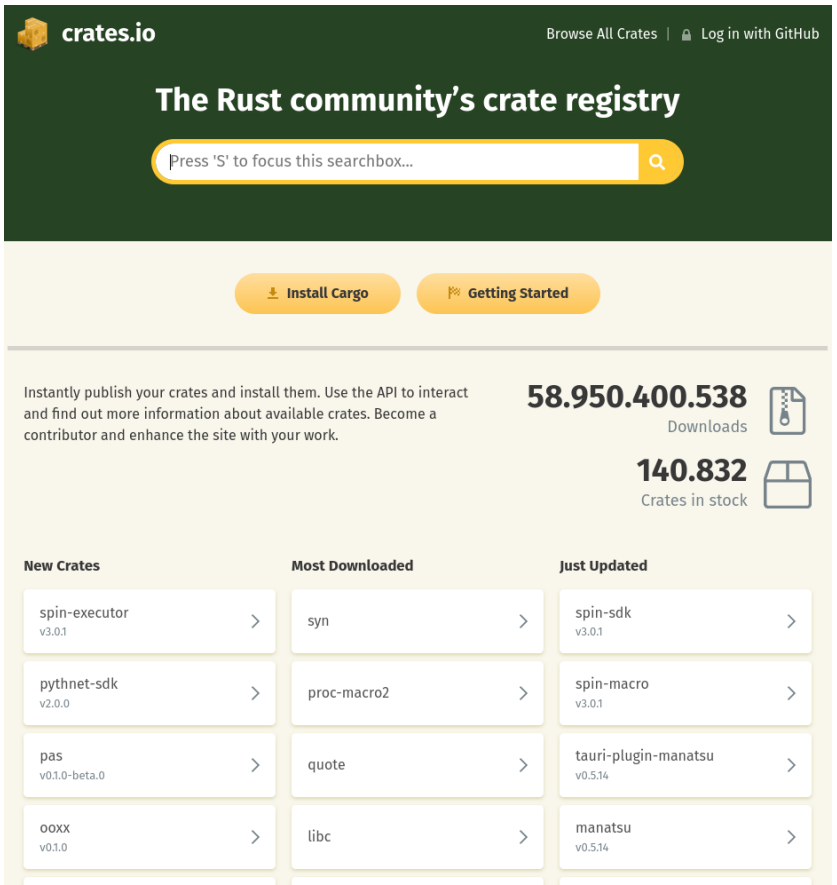
```
robge@sorpen:~/src/rust$ cargo new hello_cargo
     Created binary (application) `hello_cargo` package
robge@sorpen:~/src/rust$ cd hello_cargo/
robge@sorpen:~/src/rust/hello_cargo$ du -hs
88K     .
robge@sorpen:~/src/rust/hello_cargo$ cargo run
   Compiling hello_cargo v0.1.0 (/home/robge/src/rust/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
     Running `target/debug/hello_cargo`
Hello, world!
robge@sorpen:~/src/rust/hello_cargo$ du -hs
3,9M    .
robge@sorpen:~/src/rust/hello_cargo$ cargo clean
     Removed 21 files, 7.3MiB total
robge@sorpen:~/src/rust/hello_cargo$ du -hs
92K     .
```

- Funktion `clean` löscht alle generierten Dateien
- unklar, wie 7.3 MiB gelöscht werden können, wenn das gesamte Verzeichnis nur 3.9 MiB umfasst

Rust Package Registry (`https://crates.io`)



vom 20. 03. 2023



vom 21. 03. 2024

Befehlsübersicht

Befehl	Beschreibung
`cargo [help]`	allgemeine Hilfe zu Cargo
`cargo build, b`	übersetzt das aktuelle Package, Ergebnis in `./target`
`cargo run, r`	aktuelles Package starten (übersetzt ggf. vorher)
`cargo check, c`	übersetzt aktuelles Package, baut **nicht** das Binary
`cargo test, t`	startet definierte Tests
`cargo clean`	löscht generierte Dateien (`target/`-Verzeichnis)
`cargo update`	generiert neue `Cargo.lock`

Kritik

- ungeheure Menge Einflussfaktoren auf Build-Prozess kann binäre Reproduzierbarkeit erschweren
- Speicherplatzbedarf, Menge an erzeugten Dateien (schlank ist anders)
- `man cargo` gibt's nicht
- kein Kommando zum Löschen von Projektverzeichnissen (Asymmetrie)

Zusammenfassung

- `cargo` automatisiert viele Aspekte der Projektverwaltung (das, was man mittels `make` per Hand gemacht hat)
  1. Management von Abhängigkeiten
  2. Versionsverwaltung
  3. Build-Prozess
  4. Testen (Integration, Unit Tests)

Weiterführender Lesestoff

- „The Cargo Book“: `https://doc.rust-lang.org/cargo/`
- Ukpai Ugochi: Demystifying Cargo in Rust. 5. Mai 2021, `https://blog.logrocket.com/demystifying-cargo-in-rust/`
- Tom Preston-Werner: Tom’s Obvious Minimal Language. `https://toml.io/en/`
- Tom Preston-Werner: Semantic Versioning Specification 2.0.0 `https://semver.org/spec/v2.0.0.html`
- A Tour of Rust: `https://tourofrust.com/`