Справочник Cargo

Добро пожаловать в руководство Cargo. Это руководство даст вам все, что вам нужно знать о том, как использовать Cargo для разработки Rust проектов.

Почему существует Cargo

Cargo является инструментом, который позволяет объявлять Rust проекты, их различные зависимости и проверять, что вы всегда будете получать повторимые сборки.

Для достижения этой цели, Cargo делает четыре вещи:

- Представляет два файла метаданных с различными кусочками информации о проекте.
- Загружает и выполняет построение зависимостей вашего проекта.
- Вызывает `rustc` или другой инструмент для сборки с правильными параметрами для построения проекта.
- Вводит соглашения для упрощения работы с rust проектами.

Создание нового проекта

Чтобы начать новый проект с помощью Cargo, используйте `cargo new`:

```
$ cargo new hello_world --bin
```

Мы передаём `--bin` потому, что мы делаем двоичную программу: если мы делаем библиотеку, мы бы не задали этот параметр. Это также инициализирует новый `git`-репозиторий по умолчанию. Если вы не хотите, делать этого, передайте параметр `--vcs none`.

Давайте проверим, что Cargo сделал для нас:

```
$ cd hello_world
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 каталог, 2 файла
```

Если бы мы просто использовали н`cargo new hello_world` без флага `--bin`, то мы бы получили `lib.rs` вместо `main.rs`. На данный момент однако, это все, что нам нужно для начала работы. Во-первых давайте проверим, `Cargo.toml`:

```
[package]
```

```
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
```

Это называется **манифестом**, и содержит все метаданные, которые необходимо Cargo для того, что бы скомпилировать проект.

Вот что в `src/main.rs`:

```
fn main() {
    println!("Hello, world!");
}Run
```

Cargo создал для нас «hello world». Давайте скомпилируем его:

```
$ cargo build
   Compiling hello_world v0.1.0 (file:///path/to/project/hello_world)
```

И запустим:

```
$ ./target/debug/hello_world
Hello, world!
```

Мы также можем использовать `cargo run`, чтобы скомпилировать и запустить все за один шаг (вы не увидите строк о `компиляции`, если вы не внесли изменения с момента последней компиляции):

```
$ cargo run
   Compiling hello_world v0.1.0 (file:///path/to/project/hello_world)
   Running `target/debug/hello_world`
Hello, world!
```

Теперь вы увидите новый файл `Cargo.lock`. Он содержит информацию о наших зависимостях. Поскольку мы не имеем пока никаких зависимостей, это не очень интересно.

Как только вы будете готовы для выпуска релиза, можно использовать `cargo build --release` для компиляции файлов с включенной оптимизацией:

```
$ cargo build --release
   Compiling hello_world v0.1.0 (file:///path/to/project/hello_world)
```

`cargo build --release` помещает полученный двоичный файл в каталог `target/release` вместо `target/debug`.

Компиляция в режиме отладки является режимом по умолчанию для разработки - время компиляции короче, так как компилятор не выполняет оптимизацию, но код будет выполняться медленнее. Режим релиза занимает больше времени при компиляции, но код будет выполняться быстрее.

Работа с существующим Cargo проектом

Если вы загружаете существующий проект, который использует Cargo, то он очень легко собирается.

Во-первых получите проект откуда-то. В этом примере мы будем использовать `rand`, клонированный из репозитория на GitHub:

```
$ git clone https://github.com/rust-lang-nursery/rand.git
$ cd rand
```

Для построения используйте `cargo build`:

```
$ cargo build
   Compiling rand v0.1.0 (file:///path/to/project/rand)
```

Надо будет скачать все зависимости, а затем построить их, вместе с проектом.

Добавление зависимостей от crates.io

crates.io является центральным хранилищем сообщества, которое служит в качестве места поиска и загрузки пакетов. `cargo` настроен для использования его по умолчанию, для поиска нужных пакетов.

Библиотеки от которых зависит проект, размещенные на crates.io, добавьте в свой `Cargo.toml`.

Добавление зависимостей

Если в Вашем `Cargo.toml` пока нет раздела `[dependencies]`, добавьте его , что бы он содержал список крейтов, с указанием имени и версии, которую вы хотели бы использовать. В этом примере добавляется зависимость от крейта `time!`:

```
[dependencies]
time = "0.1.12"
```

Строка версии является требование к версии на сервере. Документы Указание зависимостей содержат дополнительные сведения о параметрах, котрые вы используете.

Если мы хотели бы также добавить зависимость от крейта `regex` , нам бы не нужно добавить `[dependencies]` для каждого крейта в списке. Вот как будет выглядеть файл `Cargo.toml` целиком, с зависимостями от крейтов `time` и `regex`:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
time = "0.1.12"
regex = "0.1.41"
```

Запусте `cargo build` вновь, и cargo будет скачивать новые зависимости и все их зависимости, скомпилирует их все и обновит `Cargo.lock`:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading memchr v0.1.5
 Downloading libc v0.1.10
 Downloading regex-syntax v0.2.1
 Downloading memchr v0.1.5
 Downloading aho-corasick v0.3.0
 Downloading regex v0.1.41
   Compiling memchr v0.1.5
   Compiling libc v0.1.10
   Compiling regex-syntax v0.2.1
   Compiling memchr v0.1.5
   Compiling aho-corasick v0.3.0
   Compiling regex v0.1.41
   Compiling hello_world v0.1.0 (file:///path/to/project/hello_world)
```

Наш `Cargo.lock` содержит точную информацию, о том, какие версии всех этих зависимостей были использованы.

Теперь если крейт `regex` обновится, то мы будем по-прежнему строить с той же версией, до тех пор, пока мы не выполним `cargo update`.

Теперь можно использовать библиотеку `regex`, с помощью `extern crate` в `main.rs`.

```
extern crate regex;

use regex::Regex;

fn main() {
    let re = Regex::new(r"^\d{4}-\d{2}-\d{2}$").unwrap();
    println!("Did our date match? {}", re.is_match("2014-01-01"));
}Run
```

Во время работы он покажет:

```
$ cargo run
     Running `target/hello_world`
Did our date match? true
```

Макет проекта

Cargo использует некоторые соглашения при размещении файлов, чтобы было легко погрузиться в новый Cargo проект:

```
.
├── Cargo.lock
├── Cargo.toml
```

```
├── benches
│   └── large-input.rs
├── examples
│   └── simple.rs
├── src
│   ├── bin
│   │   └── another_executable.rs
│   ├── lib.rs
│   └── main.rs
└── tests
    └── some-integration-tests.rs
```

- `Cargo.toml` и `Cargo.lock` хранятся в корневом каталоге вашего проекта.
- Исходный код находится в директории `src`.
- Файл библиотеки по умолчанию хранится `src/lib.rs`.
- По умолчанию, исполняемым файлом является `src/main.rs`.
- Другие исполняемые файлы могут быть размещены в `src/bin/*.rs`.
- Интеграционные тесты идут в каталоге `tests` (объединённые тесты идут в каждом файле, который они тестируют).
- Исполняемые файлы примеров идут в каталоге `examples`.
- Тесты производительности идут в каталоге `benches`.

Всё это объясняется более подробно в описании манифеста.

Cargo.toml vs Cargo.lock

`Cargo.toml` и `Cargo.lock` служат двум различным целям. Прежде чем мы поговорим о них, вкратце:

- `Cargo.toml` описывает зависимости в широком смысле и пишется Вами.
- `Cargo.Lock` содержит точные сведения о вашей зависимости. Он поддерживается Cargo и его не следует изменять вручную.

Если вы создаете библиотеку, которая будет зависеть других проектов, положите `Cargo.lock` в Ваш `список игнорирования` git. Если вы создаете исполняемый файл как средство командной строки или приложение, поместите `Cargo.lock` в `git`. Если Вам интересно, почему это так, посмотрите в разделе «Почему двоичные файлы должны хранить `Cargo.lock` в системе управления версиями, а библиотеки - нет?» в разделе FAQ.

Давайте копать немного глубже.

`Cargo.toml` является файлом **манифеста**, в котором мы можем указать кучу различных метаданных о нашем проекте. Например мы можем сказать, что мы зависим от другого проекта:

```toml
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
rand = { git = "https://github.com/rust-lang-nursery/rand.git" }
```

Этот проект имеет одну зависимость, от библиотеки `rand`. В этом случае мы заявляем, что мы полагаемся на конкретный репозиторий Git, который живет на GitHub. Поскольку мы не указан другой информации, Cargo предполагает, что мы намерены использовать последний коммит на

ветку `master` для построения нашего проекта.

Неплохо выглядит? Ну есть одна проблема: Если вы строите этот проект сегодня, а затем вы отправить копию мне, и я построить этот проект завтра, что-то плохое может случиться. Там может быть больше коммитов `rand` к этому моменту, и моя сборка будет включать новые коммиты, тогда как Ваша нет. Таким образом мы получили бы разные версии программы. Это было бы плохо, потому что мы хотим воспроизводимых результатов.

Мы могли бы решить эту проблему, поставив `rev` строки в наших `Cargo.toml`:

```
[dependencies]
rand = { git = "https://github.com/rust-lang-nursery/rand.git", rev = "9f35b8e" }
```

Теперь наши сборки будут одинаковыми. Но есть большой недостаток: Теперь мы должны сами заботиться о SHA-1 каждый раз, когда мы хотим обновить нашу библиотеку. Это утомительно и приводит к ошибкам.

Введите `Cargo.lock`. Так как он существует, нам не нужно, вручную отслеживать в точности все изменения: Cargo будет делать это за нас. Если наш есть манифест выглядит следующим образом:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
rand = { git = "https://github.com/rust-lang-nursery/rand.git" }
```

Cargo будет брать последний коммит и записывать его информацию в наш `Cargo.lock`, когда мы впервые выполняем сборку. Этот файл будет выглядеть следующим образом:

```
[root]
name = "hello_world"
version = "0.1.0"
dependencies = [
 "rand 0.1.0 (git+https://github.com/rust-lang-nursery/rand.git#9f35b8e439eeedd60b9414
c58f389bdc6a3284f9)",
]

[[package]]
name = "rand"
version = "0.1.0"
source = "git+https://github.com/rust-lang-nursery/rand.git#9f35b8e439eeedd60b9414c58f
389bdc6a3284f9"
```

Вы можете видеть, что здесь есть гораздо больше информации, включая точный номер ревизии, которую мы использовали для построения. Теперь когда Вы даете проект кому-либо другому, он будет использовать тот же SHA, даже несмотря на то, что мы не указываем в нашем `Cargo.toml`.

Когда мы готовы перейти к новой версии библиотеки, Cargo может повторно вычислить зависимости и обновить это за нас:

```
$ cargo update           # updates all dependencies
$ cargo update -p rand  # updates just “rand”
```

Он запишет новый `Cargo.lock` с новой информацией о версии. Обратите внимание, что аргументом для `cargo update` фактически является спецификации ID пакета и `rand` просто кратко специфицируется.

Тесты

Cargo может выполнять тесты с помощью команды `cargo test`. Cargo ищет тесты для выполнения в двух местах: в каждом из `src` файлов и любых тестах в каталоге `tests/`. Тесты в Ваших `src` файлах должны быть модульными тестами, а тесты в каталоге `tests/` должна быть тестами интеграции. Таким образом Вам нужно будет импортировать Ваши крейты в файлы `тестов`.

Вот пример выполняющегося `cargo test` в нашем проекте, который, в настоящее время, ничего не тестирует:

```
$ cargo test
   Compiling rand v0.1.0 (https://github.com/rust-lang-nursery/rand.git#9f35b8e)
   Compiling hello_world v0.1.0 (file:///path/to/project/hello_world)
     Running target/test/hello_world-9c2b65bbb79eabce

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Если наш проект содержит тесты, мы увидим больше выдачи с правильным количеством тестов.

Можно также запустить определенный тест путем передачи фильтра:

```
$ cargo test foo
```

Это запустит любой тест с `foo` в его названии.

`cargo test` запускает так же дополнительные проверки. Например он будет компилировать все примеры, которые Вы включили и будет также проверить примеры в документации. Пожалуйста, смотрите руководство по тестированию в документации Rust для получения более подробной информации.

Travis CI

Чтобы проверить проект на Travis CI, здесь есть пример `.travis.yml` файла:

```
language: rust
rust:
  - stable
  - beta
  - nightly
matrix:
  allow_failures:
    - rust: nightlyRun
```

Это будет тест всех трёх релизов каналов, но любые поломки в ночных релизах не будут прекращать общее построение. Пожалуйста смотрите Travis CI Rust документацию для получения дополнительной информации.

Дополнительная литература

Теперь, когда у вас есть обзор о том, как использовать cargo и создали свой первый крейт, вы можете быть заинтересованы в:

- Публикация вашего крейта на crates.io
- Чтение о всех возможных способах указания зависимостей
- Получить более подробную информацию о то, что можно указать в манифесте Вашего `Cargo.toml`

Еще больше тем доступно в меню «документы» в верхней части этой странички!