Все, что вам нужно знать о кросс компиляции Rust программ!

COLLABORATORS

	TITLE : Все, что вам нужно знать о кросс компиляции Rust программ!	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Сергей Ларионов	1 октября 2016	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
1.0	1 октября 2016		СЛ

Содержание

Rust как кросс компилятор

Если вы хотите настроить Ваш Rust toolchain как кросс-компилятор, вы пришли в нужное место! Я задокументировал все необходимые шаги, а также ошибки и общие проблемы, которые вы можете встретить на этом пути.

Уважаемый читатель, если вы заметили опечатку, битую ссылку, или плохо сформулированное/-неверное предложение/абзац, пожалуйста, откройте вопрос, указывая на проблему, и я исправлю текст. Запросы на исправление опечаток или неработающие ссылки конечно же, принимаются с радостью!

Пример на ОС Ubuntu

Вот команды, необходимые для создания и установки стабильной тулдчейн Rust как кросс-компилятор для ARMv7 [1] на новом дистрибутиве Ubuntu Trusty. Цель этого примера — показать, что кросс-компилятор прост в установке и даже ещё проще в использовании.

```
# Установите Rust. rustup.rs настоятельно рекомендуется.
# См. https://www.rustup.rs/ для подробностей
# Вы можете использовать multirust.
# См. https://github.com/brson/multirust для подробностей
$ curl https://sh.rustup.rs -sSf | sh

# Шаг 0: Целевая платформа ARMv7, триплет такой платформы: ‘armv7-unknown-linux-gnueabihf‘

# Шаг 1: Установка C кросс toolchain
$ sudo apt-get install -qq gcc-arm-linux-gnueabihf

# Шаг 2: Установить кросс-скомпилированные стандартные крейты
$ rustup target add armv7-unknown-linux-gnueabihf

# Шаг 3: Настройка cargo для кросс-компиляции
$ mkdir -p ~/.cargo
$ cat >>~/.cargo/config <<EOF
> [target.armv7-unknown-linux-gnueabihf]
> linker = ”arm-linux-gnueabihf-gcc”
> EOF

# Тест кросс-компиляции cargo-проекта
$ cargo new --bin hello
$ cd hello
$ cargo build --target=armv7-unknown-linux-gnueabihf
Компиляция hello.rs v0.1.0 (file:///home/ubuntu/hello)
$ file target/armv7-unknown-linux-gnueabihf/debug/hello
hello: ELF 32-bit LSB  shared object, ARM, EABI5 version 1 (SYSV), dynamically linked (uses ↩
     shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=67 ↩
    b58f42db4842dafb8a15f8d47de87ca12cc7de, not stripped

# Проверить двоичный файл
$ scp target/armv7-unknown-linux-gnueabihf/debug/hello me@arm:~
$ ssh me@arm:~ ./hello
Hello, world!
```

Сейчас Вы можете выполнять кросс-компиляцию

Для дополнительных примеров посмотрите построение Travis CI.

[1] На ARMv7, эти программы не будут работать для кросс компиляции для Raspberry Pi (1), так как это устройство ARMv6.

Остальная часть руководства будет объяснить и обобщать каждый шаг, выполненный в предыдущем примере. Содержание

Это руководство состоит из двух частей: «Основной текст» и дополнительные темы. Основной текст охватывает простейший случай: кросс компиляции Rust программ, которые зависят только от крейта std «поддериваемой целевой платформы», для которой доступны официальные сборки. В дополнительных темах рассматриваются вопросы no_std программ, файлы спецификации цели, как кросс-компилировать «стандартные крейты» и устранение распространенных проблем.

Дополнительные темы строятся на основе информации, которая объясняется в основном тексте. Таким образом, даже если Ваш вариант использования не является таким же, как в основном тексте, следует сначала прочитать основной текст, а потом перейти в раздел дополнительных тем.

Терминология

Давайте, прежде всего, убедимся в том, что мы говорим на одном и том же языке, путем определения некоторых терминов!

В своей самой основной форме кросс компиляция включает в себя две различные системы/компьютеры/устройства. Хост-система, где программа скомпилирована, и целевая системы, где будет выполняться скомпилированная программа.

Например если вы кросс компилируете Rust программы на вашем ноутбуке, чтобы выполнить его на Raspberry Pi 2 (RPi2). Тогда ваш ноутбук является хостом, а RPi2 является целью.

Однако (кросс) компилятор не создаёт двоичный код, который работает только на одной системе (например RPi2). Полученный двоичный файл также может быть выполнен на нескольких других системах (например ODROIDs) которые имеют некоторые характеристики, как их архитектура (например, ARM) и их операционной системы (например, Linux). Для ссылки на этот набор систем с общими характеристиками мы используем строку под названием триплет.

- Архитектура: arm.
- Производитель: неизвестно. В этом примере производитель не указан и/или это не имеет значения.
- система: linux.
- ABI: gnueabihf. gnueabihf указывает, что система использует glibc как стандартную библиотеку (libc) C и имеет аппаратное ускорение арифметики с плавающей точкой (то есть FPU).

И все системы, такие как RPi2, ODROIDs, и почти все имеющие ARMv7, на которых работает GNU/Linux, принадлежат к этому триплету.

Некоторые триплеты пропускают поставщика или abi компоненту, поэтому они на самом деле «двойки». Примером такого триплета является x86_64-apple-darwin, где:

- Архитектура: x86_64.
- Производитель: apple.
- Система: Дарвин.

Замечание
Сейчас я буду перегружать термин ”цель”, который означает одну целевую систему, а также ссылается на набор систем с общими характеристиками, предусмотренных некоторыми триплетом.

Требования

Для компиляции Rust программы нам нужны 4 вещи:

1. Узнайте триплет для целевой системы.
2. Gcc кросс-компилятор, поскольку rustc использует gcc для линковки объектов.
3. С зависимости, обычно «libc», для выполнения кросс компиляции для целевой системы.
4. Rust зависимости, обычно крейт std, кросс компилированный для целевой системы.

Целевой триплет

Чтобы узнать, триплёт для вашей цели, необходимо сначала выяснить четыре типа информации о цели:

1. Архитектура: На UNIX системах вы можете определить это командой uname -m.
2. Поставщик: На linux: обычно unknown. На windows: pc. На OSX/iOS: apple
3. Система: На UNIX системах вы можете определить это командой uname -s.
4. ABI: На Linux, это определяется реализацией libc, которую вы можете узнать с помощью вызова ldd --version. Mac и

Замечание
BSD системы не обеспечивают разные ABI, поэтому это поле опускается. На windows, насколько я знаю, есть только два ABI: gnu и msvc.

Далее вам нужно сравнить эту информацию со списком целей, поддерживаемых rustc и проверить, если есть совпадение. Если у вас есть rustc nightly-2016-02-14, 1.8.0-beta.1 или новее , вы можете использовать rust --print target-list, чтобы получить полный список поддерживаемых целей. Вот список поддерживаемых целей по состоянию на 1.8.0-beta.1:

Список целей для кросс компиляции

```
$ rustc --print target-list | pr -tw100 --columns 3
aarch64-apple-ios                i686-pc-windows-gnu              x86_64-apple-darwin
aarch64-linux-android            i686-pc-windows-msvc             x86_64-apple-ios
aarch64-unknown-linux-gnu        i686-unknown-dragonfly           x86_64-pc-windows-gnu
arm-linux-androideabi            i686-unknown-freebsd             x86_64-pc-windows-msvc
arm-unknown-linux-gnueabi        i686-unknown-linux-gnu           x86_64-rumprun-netbsd
arm-unknown-linux-gnueabihf      i686-unknown-linux-musl          x86_64-sun-solaris
armv7-apple-ios                  le32-unknown-nacl                x86_64-unknown-bitrig
armv7-unknown-linux-gnueabihf    mips-unknown-linux-gnu           x86_64-unknown-dragonfly
armv7s-apple-ios                 mips-unknown-linux-musl          x86_64-unknown-freebsd
asmjs-unknown-emscripten         mipsel-unknown-linux-gnu         x86_64-unknown-linux-gnu
i386-apple-ios                   mipsel-unknown-linux-musl        x86_64-unknown-linux-musl
i586-unknown-linux-gnu           powerpc-unknown-linux-gnu        x86_64-unknown-netbsd
i686-apple-darwin                powerpc64-unknown-linux-gnu      x86_64-unknown-openbsd
i686-linux-android               powerpc64le-unknown-linux-gnu
```

Замечание
Замечание Если вы хотите знать, в чём разница между arm-unknown-linux-gnueabihf и armv7-unknown-linux-gnueabihf, триплет arm покрывает ARMv6 и ARMv7 процессоры, тогда как armv7 поддерживает только процессоры ARMv7. По этому, триплет armv7 включает оптимизацию, которая возможна только на процессорах ARMv7. С другой стороны, если Вы используете arm триплет, Вам придется отказаться от оптимизации путем передачи дополнительных флагов, подобных - C target-feature=+Неон в rustc. Резюмируя: для ускорения работы двоичных файлов, используйте armv7 если целевой процессор ARMv7.

Если Вы не нашли триплет, который соответствует вашей целевой системе, то Вам придётся создать файл спецификации цели .

От этой точки и далее, я буду использовать термин $rustc_target для обозначения триплета, который Вы нашли в этом разделе. Например, если вы обнаружили, что ваша цель arm-unknown-linux-gnueabihf, то всякий раз, когда вы видите что-то вроде -target = $rustc_target мысленно подставьте содержимое $rustc_target, так что, в итоге Вы увидите target = arm-unknown-linux-gnueaibhf.

Аналогичным образом я буду использовать термин $host для обозначения триплета хоста. Вы можете найти этот триплет в выдаче команды rustc -Vv в поле host. Например моя хост-система имеет триплет x86_64-unknown-linux-gnu.

Кросс тулчейн для C

Здесь все становится немного запутанным.

GCC кросс-компиляторы иеют целью только один триплет. И этот триплет используется как префикс для всех команд toolchain: ar, gcc, и др. Это помогает отличить инструмент, используемый для нативной компиляции, например gcc, от средства кросс компиляции, например arm-none-eabi-gcc.

Путаница в том, что триплеты могут быть весьма произвольными, поэтому Ваш кросс компилятор C, скорее всего будет иметь префиксом триплет, который отличается от $rustc_target. Например в Ubuntu пакеты кросс-компиляторов для устройств ARM, называются как arm-linux-gnueabihf-gcc, и тот же самый кросс-компилятор имеет префикс как armv7—unknown-linux-gnueabihf-gcc в Exherbo, а rustc использует для этой цели триплёт arm-unknown-linux-gnueabihf. Ни один из этих триплетов не совпадает , но они относятся к одному набору систем.

Лучший способ убедиться, что у вас есть правильный крjcc toolchain для вашей целевой системы является кросс-компиляции C программы, желательно что-то не тривиальное и испытания её, выполнение на целевой системе.

О том, где можно получить C кросс toolchain, которая будет зависеть от вашей системы. Некоторые дистрибутивы Linux предоставляют пакеты кросс компиляторов. В других случаях необходимо скомпилировать кросс-компилятор самостоятельно. Такие инструменты, как crosstool-ng могут помочь в этом начинании. Для кросс системы на Linux для OSX, поищите в проекте osxcross.

Некоторые примеры пакетов кросс компиляторов ниже:

Для arm-unknown-linux-gnueabi, Ubuntu и Debian предоставляют gcc-* - arm - linux-gnueabi пакеты, где * — версия gcc. Пример: gcc 4.9-arm-linux-gnueabi

Для arm-unknown-linux-gnueabihf так же, как выше, но надо заменить gnueabi на gnueabihf

Для OpenWRT устройств, то есть когда цель mips-unknown-linux-uclibc (15.05 и старше) и mips-unknown-linux-musl (post 15.05), используйте OpenWRT SDK

Для Raspberry Pi используйте инструменты Raspberry.

Обратите внимание, что C кросс toolchain будет поставляться с кросс компилированной libc для вашей целевой платформы. Убедитесь, что:

libc в тулчейне соответствует целевой libc. Например, если ваша цель использует musl libc, то ваш toolchain должен также использовать musl libc.

Toolchain libc является ABI совместим с целевой libc. Обычно это означает, что toolchain libc должен быть старше, чем целевой libc. В идеале toolchain libc и целевой libc должны иметь ту же версию.

Начиная с этой точки, я буду использовать обозначение $gcc_prefix для обозначения префикса инструментов перекрестной компиляции (то есть кросс toolchain), которые Вы установили в этом разделе.

Кросс компилированный крейт Rust

Большинство rust программ используют крейт std, так что по крайней мере Вам понадобится кросс компилированный крейт std для кросс компиляции программы. Самый простой способ получить его – из официальных сборок.

Если вы используете multirust, версия от 2016-03-08, эти крейты можно установить с помощью одной команды: multirust add-target nightly $rustc_target. Если вы используете rustup.rs, используйте команду: rustup target add $rustc_target. И если Вы ничего этого не используете, следуйте инструкциям ниже, чтобы вручную установить крейты.

Архив, который Вам нужен, это $date/rust-std-nightly-$rustc_target.tar.gz. Где $date обычно соответствует rustc дате комита, котрую показывает rustc -V, хотя в некоторых случаях даты могут отличаться от одного до нескольких дней.

Например, для цели arm-unknown-linux-gnueabihf и rustc с версией (rustc -V) rustc 1.8.0-nightly (3c9442fc5 2016-02-04) это будет правильным архивом:

http://Static.Rust-lang.org/dist/2016-02-04/Rust-STD-Beta-ARM-Unknown-Linux-gnueabihf.tar.gz

Чтобы установить архив используйте скрипт install.sh, который находится внутри архива:

Получение архива rust

```
$ tar xzf rust-std-nightly-arm-unknown-linux-gnueabihf.tar.gz
$ cd rust-std-nightly-arm-unknown-linux-gnueabihf
$ ./install.sh --prefix=$(rustc --print sysroot)
```

Внимание
Эти команда будут выводить сообщение, которое выглядит следующим образом: ”creating uninstall script at /some/path/lib/rustlib/uninstall.sh”. Не запускайте его, потому что он будет удалять кросс компилированные стандартные крейты и нативные стандартные крейты; Вы останетесь с непригодной для использования установкой rust и Вы не сможете компилировать даже для нативной платформы.

Если по какой-либо причине Вам нужно деинсталлировать крейты, которые Вы только что установили, просто удалите следующий каталог: $(rustc --print sysroot)/lib/rustlib/$rustc_target.

ЗАМЕЧАНИЕ: Если Вы используете nightly channel каждый раз, когда Вы обновляете Вашу установку rust, Вам придется установить новый набор кросс-скомпилированных стандартных крейтов. Для этого просто загрузите новый архив и используйте скрипт install.sh как и раньше. Насколько я помню, скрипт будет также заботиться об удалении старого набора крейтов.

Кросс компиляция с помощью Rust

Это легкая часть!

Перекрестная компиляция с rustc требует передачи нескольких дополнительных флагов при его вызове:

```
--target=$rustc_target, говорит rustc, что мы желаем кросс компиляцию для $rustc_target.
- C  linker=$gcc_prefix-gcc, предписывает rustc использовать кросс компоновщик вместо  ↩
    нативного (cc).
```

Далее, пример для проверки правильности установки кросс компиляции:

Создайте программу hello world на хосте

```
$ cat hello.rs
fn main() {
println!(”Hello, world!”);
}
```

Кросс-компиляция программы на хосте

```
$ rustc \
--target=arm-unknown-linux-gnueabihf \
- C linker=arm-linux-gnueabihf-gcc \
hello.rs
```

Запустите программу на целевой машине

```
$ scp hello me@arm:~
$ ssh me@arm ./hello
Hello, world!
```

Кросс компиляция с помощью Cargo

Для кросс-компиляции с помощью cargo, мы должны сначала использовать его настройки системы для установки правильного компоновщика и архиватор для целевого объекта. После установки, нам нужно только передать флаг -target в команду cargo.

Конфигурация cargo хранится в файле TOML, ключ, который нас интересует, target.$rustc_target.linker Значение, хранящееся в этом ключе — то же, самое, которое мы передавали rustc в предыдущем разделе. Вы должны решить, Вы делаете эту конфигурацию глобальной, или для конкретного проекта.

Давайте рассмотрим пример:

Создайте новый двоичный cargo проект.

```
$ cargo new --bin foo
$ cd foo
```

Добавление зависимостей проекта.

```
$ echo ’clap = ”2.0.4”’ >> Cargo.toml
$ cat Cargo.toml
[package]
authors = [”me”, ”myself”, ”I”]
name = ”foo”
version = ”0.1.0”

[dependencies]
clap = ”2.0.4”
```

Настройте цели, компоновщик и архиватор только для этого проекта.

```
$ mkdir .cargo
$ cat >.cargo/config <<EOF
> [target.arm-unknown-linux-gnueabihf]
> linker = ”arm-linux-gnueabihf-gcc”
> EOF
```

Напишите приложение

```
$ cat >src/main.rs <<EOF
> extern crate clap;
>
> use clap::App;
>
> fn main() {
>     let _ = App::new(”foo”).version(”0.1.0”).get_matches();
> }
> EOF
```

Постройте проект для цели

```
$ cargo build --target=arm-unknown-linux-gnueabihf
```

Установите двоичный файл на цель

```
$ scp target/arm-unknown-linux-gnueabihf/debug/foo me@arm:~
```

Запустите программу на целевой машине

```
$ ssh me@arm ./foo -h
foo 0.1.0

USAGE:
foo [FLAGS]

FLAGS:
- h, --help       Prints help information
- V, --version    Prints version information
```

Дополнительные темы

Кросс компиляция стандартного крейта

На самом деле, Вам достаточно только кросс-компилировать стандартные крейты, если Ваша цель поддерживается системой построения rust (RBS). Вы можете найти список всех поддерживаемых целей в каталоге mk/cfg (Обратите внимание, каталог линкера не является последней редакцией). По состоянию в rustc 1.8.0-nightly (3c9442fc5 2016-02-04), я вижу следующие поддерживаемые цели:

Список поддерживаемых целей

```
$ ls mk/cfg
aarch64-apple-ios.mk              i686-pc-windows-msvc.mk          x86_64-pc-windows-gnu. ↩
    mk
aarch64-linux-android.mk          i686-unknown-freebsd.mk          x86_64-pc-windows-msvc. ↩
    mk
aarch64-unknown-linux-gnu.mk      i686-unknown-linux-gnu.mk        x86_64-rumprun-netbsd. ↩
    mk
```

```
arm-linux-androideabi.mk          le32-unknown-nacl.mk              x86_64-sun-solaris.mk
arm-unknown-linux-gnueabihf.mk    mipsel-unknown-linux-gnu.mk       x86_64-unknown-bitrig. ↩
    mk
arm-unknown-linux-gnueabi.mk      mipsel-unknown-linux-musl.mk      x86_64-unknown- ↩
    dragonfly.mk
armv7-apple-ios.mk                mips-unknown-linux-gnu.mk         x86_64-unknown-freebsd. ↩
    mk
armv7s-apple-ios.mk               mips-unknown-linux-musl.mk        x86_64-unknown-linux- ↩
    gnu.mk
armv7-unknown-linux-gnueabihf.mk  powerpc64le-unknown-linux-gnu.mk  x86_64-unknown-linux- ↩
    musl.mk
i386-apple-ios.mk                 powerpc64-unknown-linux-gnu.mk    x86_64-unknown-netbsd. ↩
    mk
i686-apple-darwin.mk              powerpc-unknown-linux-gnu.mk      x86_64-unknown-openbsd. ↩
    mk
i686-linux-android.mk             x86_64-apple-darwin.mk
i686-pc-windows-gnu.mk            x86_64-apple-ios.mk
```

Обратите внимание, если Ваша цель не поддерживается RBS, то вам нужно добавить поддержку для Вашей цели. Я не стану вдаваться в подробности добавления поддержки для новой цели, но этот PR можно использовать в качестве образца.

Замечание
Если Вы занимаетесь программированием для устройства без ОС, сборкой собственного ядра или в общем, работа с #! [no_std] код, то Вы вероятно не хотите (и вероятно не потому, что нет никакой OS) строить все стандартные крейты, но только основной крейт и другие автономные крейты. Если это ваш случай, читайте раздел про кросс компиляцию no_std кода вместо этого.

Шаги для кросс-компиляции стандартных крейтов не являются сложными, но процесс их построения занимает очень много времени потому что RBS будет создавать новый компилятор, а затем использовать его, для кросс компиляции крейтов. Надеюсь, предстоящее построение системы на основе cargo откроет возможность сделать это гораздо быстрее, позволяя вам использовать уже установленные rustc и cargo для кросс компиляции стандартных крейтов.

Возвращаясь к инструкции, сначала Вам нужно выяснить, хэш комита Вашего rustc. Это отображается в разделе вывода rustc-Vv. Например это rustc:

```
$ rustc -Vv
rustc 1.8.0-nightly (3c9442fc5 2016-02-04)
binary: rustc
commit-hash: 3c9442fc503fe397b8d3495d5a7f9e599ad63cf6
commit-date: 2016-02-04
host: x86_64-unknown-linux-gnu
release: 1.8.0-nightly

Has commit hash: 3c9442fc503fe397b8d3495d5a7f9e599ad63cf6.
```

Далее вам нужно получить исходный код rust и проверите его на точное совпадение с хэшем комита. Не пропускайте checkout или Вы закончите с крейтами, которые являются непригодными для использования с Вашим компилятором.

```
$ git clone https://github.com/rust-lang/rust
$ cd rust
$ git checkout $rustc_commit_hash
# Triple check the git checkout matches ‘rustc‘ commit hash
$ git rev-parse HEAD
$rustc_commit_hash
```

Далее мы готовим каталог построения для вне исходной сборки.

```
# Anywhere
$ mkdir build
$ cd build
$ /path/to/rust/configure --target=$rustc_target
```

configure принимает многие другие флаги конфигурации, проверьте configure -help для получения дополнительной информации. Обратите внимание, что по умолчанию, то есть без каких-либо флагов, configure подготовит полностью оптимизированные сборки.

Далее мы запускаем сборку:

```
$ make -j$(nproc)
```

Если вы натолкнётесь на ошибку во время построения:

```
make[1]: $rbs_prefix-gcc: Command not found
```

Не паникуйте!

Это происходит потому, что RBS ожидает gcc с определенным префиксом для каждой цели, но этот префикс может не совпадать с префиксом установленного кросс-компилятора. Например в моей системе, установленный кросс-компилятор armv7-unknown-linux-gnueabihf-gcc, но RBS, при построении для цели arm-unknown-linux-gnueabihf, ожидает кросс-компилятор под названием arm-none-gnueabihf-gcc.

Это может быть легко исправлено с помощью некоторых двоичных файлов ”прокладок”:

```
# In the build directory
$ mkdir .shims
$ cd .shims
$ ln -s $(which $gcc_prefix-ar) $rbs_prefix-ar
$ ln -s $(which $gcc_prefix-gcc) $rbs_prefix-gcc
$ cd ..
$ export PATH=$(pwd)/.shims:$PATH
```

Теперь вы в состоянии записать как $gcc_prefix-gcc так и $rbs_prefix-gcc. Например:

```
# My installed cross compiler
$ armv7-unknown-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=armv7-unknown-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/usr/x86_64-pc-linux-gnu/libexec/gcc/armv7-unknown-linux-gnueabihf ↩
    /5.3.0/lto-wrapper
Target: armv7-unknown-linux-gnueabihf
Configured with: (...)
Thread model: posix
gcc version 5.3.0 (GCC)

# The cross compiler that the RBS expects, which is supplied by the .shims directory
$ arm-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=armv7-unknown-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/usr/x86_64-pc-linux-gnu/libexec/gcc/armv7-unknown-linux-gnueabihf ↩
    /5.3.0/lto-wrapper
Target: armv7-unknown-linux-gnueabihf
Configured with: (...)
Thread model: posix
gcc version 5.3.0 (GCC)
```

Теперь можно продолжить построение с использованием make -j$(nproc).

Надеюсь, построение завершится успешно и ваш кросс компилированные крейты будут доступны в каталоге $host/stage2/lib/rustlib/$ rustc_target/lib.

```
# In the build directory
$ ls x86_64-unknown-linux-gnu/stage2/lib/rustlib/arm-unknown-linux-gnueabihf/lib
liballoc-db5a760f.rlib           librand-db5a760f.rlib            stamp.arena
liballoc_jemalloc-db5a760f.rlib  librbml-db5a760f.rlib            stamp.collections
liballoc_system-db5a760f.rlib    librbml-db5a760f.so              stamp.core
libarena-db5a760f.rlib           librustc_bitflags-db5a760f.rlib  stamp.flate
libarena-db5a760f.so             librustc_unicode-db5a760f.rlib   stamp.getopts
libcollections-db5a760f.rlib     libserialize-db5a760f.rlib       stamp.graphviz
libcompiler-rt.a                 libserialize-db5a760f.so         stamp.libc
libcore-db5a760f.rlib            libstd-db5a760f.rlib             stamp.log
libflate-db5a760f.rlib           libstd-db5a760f.so               stamp.rand
libflate-db5a760f.so             libterm-db5a760f.rlib            stamp.rbml
libgetopts-db5a760f.rlib         libterm-db5a760f.so              stamp.rustc_bitflags
libgetopts-db5a760f.so           libtest-db5a760f.rlib            stamp.rustc_unicode
libgraphviz-db5a760f.rlib        libtest-db5a760f.so              stamp.serialize
libgraphviz-db5a760f.so          rustlib                          stamp.std
liblibc-db5a760f.rlib            stamp.alloc                      stamp.term
liblog-db5a760f.rlib             stamp.alloc_jemalloc             stamp.test
liblog-db5a760f.so               stamp.alloc_system
```

Следующий раздел расскажет вам как установить эти крейты в директории Вашей установки rust.

Установка кросс компилированных стандартных крейтов

Во-первых мы должны более внимательно взглянуть на Ваш каталог установки rust, который Вы можете получить с rustc --print sysroot:

```
# Я использую rustup.rs, Вы получите другой путь, если Вы использовали rustup.sh или  ↩
    менеджер пакетов
#  Вашего дистрибутива  для установки rust
$ tree -d $(rustc --print sysroot)
~/.multirust/toolchains/nightly
├── bin
├── etc
│   └── bash_completion.d
├── lib
│   └── rustlib
│       ├── etc
│       └── $host
│           └── lib
└── share
├── doc
│   └── (...)
├── man
│   └── man1
└── zsh
└── site-functions
```

Видите каталог lib/rustlib/$host? Вот, где хранятся ваши собственные крейты. Кросс компилированные крейты должны быть установлены рядом с этой директории. Следуя примеру из предыдущего раздела, следующая команда будет копировать стандартные крейты, построенный RBS в нужное место.

```
# In the 'build' directory
$ cp -r \
$host/stage2/lib/rustlib/$target
$(rustc --print sysroot)/lib/rustlib
```

И наконец мы проверяем, что крейты находятся в нужном месте.

```
$ tree $(rustc --print sysroot)/lib/rustlib
/home/japaric/.multirust/toolchains/nightly/lib/rustlib
├── (...)
├── uninstall.sh
├── $host
│   └── lib
│       ├── liballoc-fd663c41.rlib
│       ├── (...)
│       ├── libarena-fd663c41.so
│       └── (...)
└── $target
└── lib
├── liballoc-fd663c41.rlib
├── (...)
├── libarena-fd663c41.so
└── (...)
```

Таким образом, Вы можете установить крейты для многих целей, которые Вы хотите. Чтобы «деинсталлировать» крейты, просто удалить каталог $target.

Файлы спецификации цели

Файл спецификации цели является JSON файлом, который содержит подробные сведения о цели для rust компилятора. Этот файл спецификации имеет пять обязательных полей и несколько факультативных. Все его ключи являются строками и его значения являются строками или логическими значениями. Ниже приведен минимальный файл спецификации цели для микроконтроллеров Cortex M3:

```
{
”0”: ”NOTE: Я буду использовать эти «числовое» поля как комментарии, но они не должны ↩
    появляться в настоящих файлах»,
”1”: «Следующие пять полей являются _required_»,
”arch”: ”arm”,
”llvm-target”: ”thumbv7m-none-eabi”,
”os”: ”none”,
”target-endian”: ”little”,
”target-pointer-width”: ”32”,

”2”: «Эти поля являются необязательными. Не все возможные необязательные поля перечислены ↩
    здесь, хотя»,
”cpu”: ”cortex-m3”,
”morestack”: false
}
```

Список всех возможных ключей и их влияние на компиляцию можно найти в файле src/librustc_back/ta

Замечание
Связанный файл не является последней редакцией.

Существует два способа передать эти файлы спецификации целевых объектов для rustc, первый - передать полный путь через флаг ---target.

```
$ rustc --target path/to/thumbv7m-none-eabi.json (...)
```

Другой — просто передать «сорт файла » в --target, но тогда сам файл должен быть в рабочем каталоге или в каталоге, заданном переменной RUST_TARGET_PATH.

```
# Файл спецификации цели находится в рабочем каталоге
$ ls thumbv7m-none-eabi.json
thumbv7m-none-eabi.json

# Используем передачу «сорта файл»
$ rustc --target thumbv7m-none-eabi (...)
```

Кросс компиляция не стандартного кода

При работе с нестандартным кодом, Вам нужно только несколько автономных крейтов, в качестве ядра, и Вы, вероятно, работаете с пользовательской целью, например Cortex-M микроконтроллер, для которой не поддерживается никаких официальных билдов и Вы не можете собрать эти крейты, используя RBS.

Простое решение, позволяющее получить кросс компилированный крейт ядра, заключается в том, что сделать Вашу программу/крейт зависимой от rust крейта libcore. Это заставит cargo построить ядерный крейт как часть процесса построения cargo. Однако этот подход имеет две проблемы:

Вирусность: Вы не можете сделать ваш крейт зависимым от другого no_std крейта, если этот крейт также зависит от libcore rust.

Если вы хотите, чтобы Ваш крейт зависел от другого стандартного крейта, то необходимо создать новый крейт rust-lib$crate.

Альтернативное решение, которое не имеет этих проблем, заключается в использовании «sysroot» который содержит все кросс компилированные крейты. Я реализую этот подход в xargo. Для получения более подробной информации посмотрите репозиторий.

Устранение распространенных неполадок

Все, что может пойти не так, пойдет неправильно - Закон Мерфи

Этот раздел: Что делать когда дела идут плохо.

Не удается найти крейт

Симптом

$ cargo build --target $rustc_target error: can’t find crate for `$crate`

Причина

rustc не может найти кросс компиллированный стандартный крейт $crate в Вашей директории установки rust.

Решение

Проверьте установки кросс компилированного раздела стандартных крейтов и убедитесь, что кросс компилированный крейт $crate находится в нужном месте.

Крейт несовместим с данной версией rustc

Симптом

$ cargo build --target $rustc_target error: the crate `$crate` has been compiled with rustc $version-$channel ($hash $date), which is incompatible with this version of rustc

Причина

Версия кросс скомпилированных стандартных крейтов, которые Вы установили не соответствуют Вашей версии rustc.

Решение

Если Вы работали на nightly channel и установил официальный билд, то Вы, вероятно, получили неправильную дату tarball. Попробуйте другую дату.

Если Вы кросс компилируете крейты из исходов, то Вы скачали не тот коммит исходов. Вы будете должны построить крейты снова, но убедившись, что Вы забрали из репозитория в правильный коммит (он должен соответствовать полю commit-hash в выводе rustc - Vv).

Неопределенная ссылка

Симптом

$ cargo build --target $rustc_target /path/to/some/file.c:$line: undefined reference to `$symbol`

Причина

Сценарий выглядит следующим образом:

Стандартные крейты были кросс компилированы с помощью C кросс toolchain «A». Затем Вы кросс компилировали rust программы с использованием C кросс-toolchain «B», эта программа также связан с стандартными крейтами, полученными в предыдущем шаге.

Проблема возникает, когда компонент libc toolchain «A» новее, чем libc компонент toolchain «B». В этом случае стандартные крейты, которые кросс компилированы с «A» могут зависеть от символов libc, которые не доступны в libc «B».

Эта ошибка также будет происходить, если «A»-libc отличается от «B»-libc. Пример: toolchain «A» mips-linux-gnu и toolchain «Б» mips-linux-musl.

Решение

Если вы наблюдаете это сообщение в официальной сборке, то это ошибка. Это означает, что разработчики rust должны понизить версию компонента libc C кросс toolchain, которую они используют для создания стандартных крейтов.

Если Вы кросс компилировали стандартные крейты самостоятельно, то было бы идеально, если бы Вы использовали ту же C кросс toolchain, что и при построении стандартных крейтов и кросс компиляции rust программ.

can’t load library

Симптом

On target $ ./hello ./hello: can’t load library *libpthread.so.0*

Причина

В целевой системе отсутствует общая библиотека. Вы можете подтвердить это с ldd:

Or `LD_TRACE_LOADED_OBJECTS=1 ./hello` on uClibc-based OpenWRT devices $ ldd hello libdl.so.0 ⇒ /lib/libdl.so.0 (0x771ba000) libpthread.so.0 ⇒ not found libgcc_s.so.1 ⇒ /lib/libgcc_s.so.1 (0x77196000)

libc.so.0 ⇒ /lib/libc.so.0 (0x77129000) ld-uClibc.so.0 ⇒ /lib/ld-uClibc.so.0 (0x771ce000) libm.so.0 ⇒ /lib/libm.so.0 (0x77103000)

Все недостающие библиотеки помечаются как «не найдена».

Решение

Установите недостающие разделяемые библиотеки в целевой системе. Продолжая предыдущий пример:

target system is an OpenWRT device $ opkg install libpthread $ ./hello Hello, world!

$symbol not found

Симптом

On target $ ./hello rustc: /path/to/$c_library.so: version '$symbol' not found (required by /path/to/$rust_libr

Причина

ABI не соответствует в библиотеке, которая была динамически связана с двоичным файлом во время кросс-компиляции и библиотекой, которая устанавлена в целевой системе.

Решение

Обновление/изменение библиотеки на хост или целевой системе так, чтобы сделать их совместимы по ABI . В идеале хост и целевой объект должны иметь одинаковую версию библиотеки.

ВНИМАНИЕ: когда я говорю о библиотеке на хосте, я имею в виду кросс скомпилированную библиотеку такую, что $prefix_gcc-gcc ссылается на Вашу rust программу. Я не имею в виду нативную библиотеку, которая может быть установлена на хосте.

illegal instruction

Симптом

on target $ ./hello Illegal instruction

Причины

Замечание
Вы также можете получить сообщение об ошибке «illegal instruction», если ваша программа достигает состояния Out Of Memory (OOM). В некоторых системах, вы увидите, дополнительно сообщение «fatal runtime error: out of memory», когда Вы натыкаетесь на OOM. Если Вы уверены, что это не Ваш случай, то это проблема кросс компиляции.

Это происходит потому, что программа содержит инструкцию, которая не поддерживается целевой системой. Среди возможных причин этой проблемы, у нас есть:

Вы компилировали для аппаратной поддержки float на целевой платформе, например arm-unknown-linux-gnueabihf, но Ваша цель не поддерживает такие операции, а на самом деле там программная поддержка float, например arm-unknown-linux-gnueabi. Решение Используйте правильный триплет, в этом примере: arm-unknown-linux-gnueabi.

Вы используете правильный триплет с программной реализацией float, например arm-unknown-linux-gnueabi, для вашей целевой системы. Но Ваша C кросс toolchain была скомпилирована с аппаратной поддержкой float, что привело к тому, что float инструкции попали в двоичный файл. Решение Получите правильный toolchain, который был построен с программной поддержкой float. Подсказка: Посмотрите на флаг --with-float в выводе $gcc_prefix-gcc - v.

ЧАВо

Я хочу построить двоичные файлы для Linux, Mac и Windows.

Как выполнить кросс компиляцию с Linux на Mac?

Краткий ответ: Неизвестно.

Трудно найти кросс C toolchain (и кросс скомпилированных библиотек C) между различными ОС (за исключением, возможно с Linux на Windows). Способ, который гораздо проще и меньше подвержен ошибкам, заключается в нативном построении кода для этих целей, потому что у них только одна платформа. Вы не можете иметь прямой доступ ко всем таким ОС, но это не проблема, потому что вы можете использовать сервисы CI, такие, как Travis CI и AppVeyor. Посмотрите мой проект rust-everywhere для получения инструкций о том, как это сделать.

Как компилировать полностью статические rust бинарники?

Краткий ответ: cargo build --target x86_64-unknown-linux-musl

Для целей вида **-**-linux-gnu*, rustc всегда производит двоичные файлы, динамически связанные с glibc и другими библиотеками:

$ cargo new --bin hello $ cargo build --target x86_64-unknown-linux-gnu $ file target/x86_64-unknown-linux-gnu/debug/hello target/x86_64-unknown-linux-gnu/debug/hello: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /usr/x86_64-pc-linux-gnu/lib/ld-linux-x86-64.so.2, for GNU/Linux 2.6.34, BuildID[sha1]=a3fa7281e9ded30372b5131a2feb6f1e78a6f1cd, not stripped $ ldd target/x86_64-unknown-linux-gnu/debug/hello linux-vdso.so.1 (0x00007fff58bf4000) libdl.so.2 ⇒ /usr/x86_64-pc-linux-gnu/lib/libdl.so.2 (0x00007fc4b2d3f000) libpthread.so.0 ⇒ /usr/x86_64-pc-linux-gnu/lib/libpthread.so.0 (0x00007fc4b2b22000) libgcc_s.so.1 ⇒ /usr/x86_64-pc-linux-gnu/lib/libgcc_s.so.1 (0x00007fc4b290c000) libc.so.6 ⇒ /usr/x86_64-pc-linux-gnu/lib/libc.so.6 (0x00007fc4b2568000) /usr/x86_64-pc-linux-gnu/lib/ld-linux-x86-64.so.2 (0x00007fc4b2f43000) libm.so.6 ⇒ /usr/x86_64-pc-linux-gnu/lib/libm.so (0x00007fc4b2272000)

Для получения статически связанных двоичных файлов, Rust предоставляет две цели: x86_64-unknown-linux-musl и i686-unknown-linux-musl. Двоичные файлы, создаваемые по этим целям связаны статически с MUSL библиотекой C. Пример ниже:

$ cargo new --bin hello $ cd hello $ rustup target add x86_64-unknown-linux-musl $ cargo build --target x86_64-unknown-linux-musl $ file target/x86_64-unknown-linux-musl/debug/hello target/x86_64-unknown-linux-musl/debug/hello: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=759d41b9a78d86bff9b6529d12c8fd6b934c0088, not stripped $ ldd target/x86_64-unknown-linux-musl/debug/hello not a dynamic executable

Лицензия

Лицензировано под любой из

- Apache License, Version 2.0 (LICENSE-APACHE or http://www.apache.org/licenses/LICENSE-2.0)
- MIT license (LICENSE-MIT or http://opensource.org/licenses/MIT)

по вашему желанию.

Вклад

Если Вы явно не предусмотрели иного, то любой вклад, намеренно переданный для включения в эту работу, как это определено в лицензии Apache 2.0, должен быть с двойной лицензией как указано выше, без каких-либо дополнительных условий или условий.