Знакомство с Anaconda: что это такое и как установить

Простое руководство по Anaconda и его установке на Ubuntu 16.04 (64-bit).

Что такое Anaconda?

Перед тем, как изучать Anaconda, рассмотрим Conda.

Цитируем определение Conda с официального блога:

Conda—это менеджер пакетов с открытым кодом и система управления средой, которая работает на Windows, macOS и Linux.

Conda проста в установке, выполнении и обновлении пакетов и зависимостей. Conda легко создает, сохраняет, загружает и переключается между средами на локальном компьютере.

*Она задумывалась для программ на **Python, но может** создавать пакеты и дистрибутивы программного обеспечения **на любом языке**.*

Возникает вопрос: почему вдруг речь зашла о Conda? Все мы знаем, что это система управления пакетами, которая используется для установки и управления пакетов приложений, написанных на Python.

Система имеет и свои ограничения. Ей можно пользоваться только для пакетов Python.

`pip` работает с Python и пренебрегает зависимостями из не-Python библиотек (HDF5, MKL, LLVM), в исходном коде которых отсутствует файл установщика.

Проще говоря, `pip` – это менеджер пакетов, который облегчает установку, обновление и удаление **пакетов Python**. Он работает с виртуальными средами **Python**.

`Conda` – это менеджер пакетов для **любого программного обеспечения** (установка, обновление, удаление). Он работает с виртуальными **системными** средами.

Conda—это инструмент для управления пакетами и установщик с куда большим функционалом, чем в `pip` . Conda может обрабатывать зависимости библиотек *вне* пакетов Python, а также сами пакеты Python.

Кроме того, Conda создает виртуальную среду.

Как возникла Anaconda?

Conda написан на чистом Python, что облегчает его использование в виртуальных средах Python. Кроме того, Conda подходит для библиотек C, пакетов R, Java и т.д.

Он устанавливает двоичные системы. Инструмент `conda build` создает пакеты из исходного кода, а `conda install` выполняет установку из пакетов сборки Conda.

Conda является менеджером пакетов для Anaconda—дистрибутива Python, предоставляемого Continuum Analytics. Емкое описание Anaconda следующее:

Anaconda—это дистрибутивы Python и R. Он предоставляет все необходимое для решения задач по анализу и обработке данных (с применимостью к Python).

Anaconda—это набор бинарных систем, включающий в себя Scipy, Numpy, Pandas и их зависимости.

Scipy—это пакет статистического анализа.

Numpy—это пакет числовых вычислений.

Pandas—уровень абстракции данных для объединения и преобразования данных.

Anaconda полезна тем, что объединяет все это в единую систему.

Двоичная система Anaconda—это установщик, который собирает все пакеты с зависимостями внутри вашей системы.

Простая установка

Установка файлов иногда превращается в сущий ад. Но Anaconda куда проще, чем кажется. Я предпочитаю Ubuntu, поскольку здесь установка зависит от выполнения пары команд и хорошего сетевого подключения. Поэтому все становится еще проще. Вот дальнейшие шаги для установки Anaconda.

(Данный процесс подойдет только для 64-битных компьютеров).

Шаг 1: скачивание bash-скрипта Anaconda
Скачать последнюю версию bash-скрипта установщика Anaconda можно с официального сайта. Это можно сделать через выполнение команды curl. Если в вашей системе не установлен curl, то скачайте его через следующую команду.

```
sudo apt-get update

sudo apt-get install curl
```

Перейдите в папку /tmp.

```
cd /tmp
```

После установки curl выполните следующую команду:

```
curl -O https://repo.continuum.io/archive/Anaconda3-4.3.1-Linux-x86_64.sh
```

Размер файла—порядка 500 МБ, поэтому установка обычно занимает несколько минут. Пожалуйста, дождитесь полного скачивания файла.

Процесс установки Anaconda

Этот скриншот был сделан после скачивания скрипта. Убедитесь в стабильности сетевого подключения. В противном случае могут возникнуть ошибки при скачивании.

Шаг 2: проверка целостности
Для проверки целостности данных установщика воспользуемся криптографическим алгоритмом хеширования под названием SHA-2 (алгоритм безопасного хеширования).

```
sha256sum Anaconda3-4.3.1-Linux-x86_64.sh
```

Контрольная сумма генерируется следующей строкой после выполнения команды.

Проверка целостности данных через контрольную сумму

Шаг 3: запуск bash-скрипта
Мы почти закончили. Пакет загрузился. Теперь осталось запустить скрипт через нужную команду.

```
bash Anaconda3-4.3.1-Linux-x86_64.sh
```

На стандартном этапе проверки у вас спросят, хотите ли вы установить Anaconda. Для продолжения установки введите `yes`.

После запуска bash-скрипта

Шаг 4: установка криптографических библиотек

Это часть предыдущего процесса. Установщик спрашивает у пользователя, хочет ли он установить все криптографические библиотеки. Введите `yes` и можете продолжать. Ориентируйтесь по скриншоту ниже – вы увидите примерно ту же информацию.

Криптографические библиотеки

Шаг 5: подтверждение папки

Последним и итоговым шагом является подтверждение папки, куда будут выгружаться все пакеты Anaconda. Укажите путь, нажмите Enter и готово! Anaconda начнет творить чудеса, устанавливая все, что вам нужно!

Установка пути для пакетов Anaconda

Шаг 6: активация и проверка
Для активации установки нужно получить файл `~/.bashrc` через следующую команду:

```
source ~/.bashrc
```

Проверяем установку через команду `conda .`

```
conda list
```

Вы увидите данные по всем пакетам, доступным с установкой Anaconda.

Перевод статьи Nandhini Saravanan: An introduction to Anaconda: what it is, and how to install it

---

Кросс-компиляция программ Rust для запуска на маршрутизаторе

Вы никогда не задумывались о том, чтобы запустить на домашнем роутере какой-нибудь пакет Ubuntu? Для этого можно было бы использовать контейнеры LXC. Всё это легко настроить, но такая установка будет не очень эффективной: зачем устанавливать целый дистрибутив Linux на маршрутизатор ради маленьких программ, используемых в домашней автоматизации?

В качестве примера в статье рассматривается маршрутизатор Turris Omnia.

Какие ещё есть варианты?

- Писать программы на Python. Turris OS реализована на версии Python 2.7, и многие инструменты администрирования также написаны на Python. Я

никогда не испытывал особого интереса по отношению к Python, к тому же это высокоуровневый интерпретируемый язык со сборщиком мусора. В общем, не самый эффективный вариант.

- Использовать низкоуровневый язык типа C или C++ и выполнять компиляцию программ в код целевой платформы для Turris. Оказывается, кросс-компиляцию довольно легко настроить, и мы можем получить оптимальную производительность при самом эффективном использовании памяти. Всё это так, если бы только не было регулярной утечки памяти или аварийного завершения работы процесса при обращении к памяти, которая уже была освобождена (не говоря уже о сложности установки сторонних зависимостей из-за отсутствия диспетчера пакетов для C/C++).
- Использовать Rust, чтобы взять максимум из этих двух вариантов: продуктивность и надежность высокоуровневых языков, таких как Python, вместе с производительностью низкоуровневых языков типа C.

Остановимся на варианте с Rust. Чтобы получить программу Rust, запускаемую на Turris, нужно:

1. Найти платформу, используемую оборудованием маршрутизатора. Она будет целью кросс-компиляции.
2. Установить инструментальные средства кросс-компиляции и убедиться, что мы можем выполнить компиляцию простой программы на C в код целевой платформы и запустить её на маршрутизаторе.
3. Установить Rust, настроить кросс-компиляцию и убедиться, что мы можем выполнить компиляцию простой программы на Rust в код целевой платформы и запустить её на маршрутизаторе.

Примечание: следующие инструкции предназначены для Ubuntu 18.04 LTS. Для запуска Ubuntu на компьютере с Windows я использую подсистему Windows для Linux.

Находим целевую платформу

Цели кросс-компиляции обычно представлены в таком формате:

```
{architecture}-{vendor}-{system}-{abi}
```

Мы уже знаем, что процессор Turris Omnia имеет архитектуру ARMv7. В случае с системами Linux поставщик обычно неизвестен (`unknown`), потому что имя поставщика дистрибутива не так важно. Последняя часть (ABI)—это двоичный интерфейс приложений, используемый операционной системой Turris OS. В Linux это связано с реализацией libc, которую можно узнать с помощью `ldd --version`.

```
$ ssh root@192.168.1.1 "ldd --version"
musl libc (armhf)
Version 1.1.19
Dynamic Program Loader
Usage: ldd [options] [--] pathname
```

Цель кросс-компиляции для Turris Omnia представлена в таком виде: `armv7-unknown-linux-musleabihf` . К счастью для нас, в Rust есть поддержка 2-ого уровня платформ:

Второй уровень платформ предусматривает«гарантированную сборку». Автоматические тесты не выполняются, поэтому создание рабочей сборки не

гарантируется, но платформы работают обычно довольно хорошо, и предлагаемые улучшения всегда приветствуются!

Настраиваем кросс-компилятор на C

Прежде чем начинать, давайте убедимся, что у нас установлен необходимый инструментарий.

```
$ sudo apt-get install build-essential
```

Наша задача стала бы намного проще, если бы мы нацеливались на `gnueabihf` вместо `musleabihf`, потому что в Ubuntu есть пакеты с набором инструментальных средств кросс-компиляции для целей `gnueabihf`. MUSL пока что для меня загадка: я почти ничего не знаю об этой реализации стандартной библиотеки C.

К счастью, есть проект под названием musl-cross-make, который даст нам *«простую makefile-сборку для кросс-компилятора musl»*, и он отлично работает! Сначала загружаем исходные коды с GitHub:

```
$ wget https://github.com/richfelker/musl-cross-make/archive/master.tar.gz
$ tar xzf master.tar.gz
$ cd musl-cross-make-master
```

Давайте настроим параметры конфигурации, прежде чем приступать к сборке. Скопируем файл шаблона конфигурации `config.mak.dist` в `config.mak` и установим следующие параметры (вы можете отменить преобразование соответствующих строк шаблона в комментарий):

```
TARGET=arm-linux-musleabihf
OUTPUT=/usr/local
MUSL_VER = 1.1.19
```

(Лучше использовать ту же версию MUSL, которая указана в `ldd --version` на вашей Turris. У меня на момент написания статьи была версия `1.1.19`).

Пора делать сборку!

```
$ make
$ make install
```

Весь инструментарий у нас должен быть установлен в `/usr/local` и доступен на `PATH`. Давайте проверим и запустим gcc:

```
$ arm-linux-musleabihf-gcc --version
arm-linux-musleabihf-gcc (GCC) 9.2.0
Copyright (C) 2019 Free Software Foundation, Inc.
(...)
```

Выполняем кросс-компиляцию программы на C

Пока что у нас есть только предположительная цель компиляции, соответствующая платформе Turris. Пришло время проверить наши

предположения на практике.

Напишем простую программу “Hello world” на C и сохраним следующий код в файл с именем `hello.c` :

```
#include <stdio.h>
int main() {
   printf("Hello, World!\n");
   return 0;
}
```

Выполняем кросс-компиляцию этой программы для Turris:

```
arm-linux-musleabihf-gcc hello.c -o hello
```

Загружаем программу на маршрутизатор и выполняем её там. Я сохраняю файл в `/srv` (который поддерживается накопителем mSATA SSD), чтобы избежать ненужных записей во внутреннюю флэш-память. Если всё пройдёт хорошо, вы должны получить знакомое приветствие:

```
$ scp hello root@192.168.1.1:/srv
hello                                 100% 7292     1.6MB/s   00:00
$ ssh root@192.168.1.1 /srv/hello
Hello, World!
```

Устанавливаем Rust и настраиваем кросс-компиляцию

Есть разные способы установки Rust, я остановил свой выбор на рекомендуемом подходе, в котором используется `rustup` .

```
$ curl https://sh.rustup.rs -sSf | sh
```

Необходимо также установить стандартные крейты (базовые модули Rust), скомпилированные для нашей целевой платформы.

```
$ rustup target add armv7-unknown-linux-musleabihf
```

На последнем этапе мы сообщаем компилятору Rust, какой компоновщик следует использовать при компиляции для целевой платформы. Добавляем в `~/.cargo/config` следующее:

```
[target.armv7-unknown-linux-musleabihf]
linker = "arm-linux-musleabihf-gcc"
```

Выполняем компиляцию программы на Rust в код целевой платформы

Создаём программу “Hello world” на Rust и компилируем её:

```
$ cargo new --bin hello
$ cd hello
$ cargo build --target=armv7-unknown-linux-musleabihf
```

```
    Compiling hello v0.1.0 (/home/bajtos/src/hello)
     Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

Загружаем программу на маршрутизатор и выполняем её там. Если всё идёт хорошо, снова получаем наше приветствие.

```
$ scp target/armv7-unknown-linux-musleabihf/debug/hello root@192.168.1.1:/srv
hello                                 100% 2903KB  10.8MB/s   00:00 $ ssh root@129.168.1.1 /s
Hello, World!
```

Поздравляю, теперь вы можете взять любую программу на Rust и запустить её на маршрутизаторе Turris Omnia. Например, если вы используете Mastodon и Twitter, то можете синхронизировать между этими двумя сетями всё то, что вы в них выкладываете, с помощью mastodon-twitter-sync.

Полезные ссылки и благодарности

Большая часть информации в этой статье основана на замечательном руководстве по кросс-компиляции программ на Rust.

Список платформ, поддерживаемых Rust, можно найти в официальной документации проекта здесь.

И наконец, мне потребовалась бы целая вечность, чтобы понять, как выполняется кросс-компиляция для MUSL ABI, если бы не было отличного проекта musl-cross-make project Рича Фелкера.

P.S.

Вы заметили, что наша программа на C весит 7 Кб, в то время как версия Rust имеет 2903 Кб? Есть несколько способов уменьшить размер исполняемого файла программ на Rust. Активировав оптимизацию во время компоновки, мне удалось быстро уменьшить размер полученной сборки до 1408 Кб. О других, более продвинутых способах можно узнать в *«Минимизации размера двоичного кода Rust»*: https://github.com/johnthagen/min-sized-rust.

Читайте также:

- Rust и разработка кроссплатформенных решений для мобильных устройств
- Изучаем WebAssembly с помощью Rust
- Использование строк в Rust

Перевод статьи Miroslav Bajtoš: Cross-compile Rust programs to run on Turris Omnia