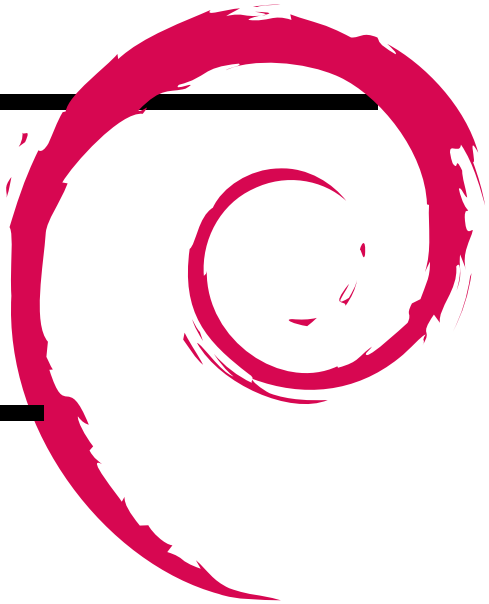
第 144 回 関西 Debian 勉強会資料

関西 Debian 勉強会担当者 佐々木・倉敷・のがた・かわだ・おおつき

2019 年 3 月 24 日

1 最近の Debian 関係のイベント報告

Debian JP



2　事前課題

関西 Debian 勉強会

参加者の皆さんは以下の通りです:

2.1　Katsuki Kobayashi

3 Rust で書いたツールの debian パッケージングに挑戦してみた話(仮)

小林 克希

3.1 はじめに

今回の内容ですが、最近個人的に触りだした Rust というプログラミング言語について、そもそもどういう言語なんだというご紹介 (といっても、私も勉強中の身ですが) と、Rust で書いたツールの deb を作ってみる、というところまでを目標としています。

3.2 Rust とは

Rust とは、Mozilla がブラウザエンジン Servo のために開発したシステムプログラミング言語です。2018 年のスタックオーバーフローの調査[*1]で愛されている言語ランキング 1 位になるなど、最近では結構メジャーになっていている感じの言語です。特徴としては、安全性・並行性について考えられて設計されている点で、かつ、みんな大好き (?) 静的型付け言語です。

なお、Rust プログラマーの事を rustacean(ラストシアン) というようです。これは、“Crustacean(甲殻類)“ から来ているらしく、そのせいか、オライリー本の表紙はオオヒロバオウギガニというカニですし、非公式マスコットもかわいらしいカニ (名前は Ferris[*2]) になってます。

3.2.1 Rust のツールチェーンをインストール

ではさっそく Rust のツールチェーンをインストールしてみましょう。さて、もちろん Rust のツールチェーンの debian パッケージはありますし、ローカルマシンにインストールせずにブラウザで色々試せるサイト[*3] もあるのですが、Rust は 2018 年の年末に 2018 Edition[*4]という大きめな改版があったので、ひとまず Rust 公式の手段でツールチェーンをインストールしましょう。例によって例のごとく、皆様の大事なホームディレクトリに色々と入れてしまいますが、バージョンアップの頻度はそれなりにあるので、Rust の最新を追っていくなら、無理せずに公式のツールチェーンを入れてしまった方が無難かと思います。

Rust のツールチェーンですが、以前はそうでもなかったみたいですが、最近では rustup[*5]というインストーラー

[*1] https://insights.stackoverflow.com/survey/2018
[*2] http://rustacean.net/
[*3] https://play.rust-lang.org/
[*4] https://blog.rust-lang.org/2018/12/06/Rust-1.31-and-rust-2018.html
[*5] https://rustup.rs/

を使うのが公式手順のようです。いかにも最近のツールっぽいですが、以下のように curl コマンドを叩いてインストールします。

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

rustup を使うと、バイナリは ~/.cargo 以下にインストールされます。パスを通したい場合、 ~/.cargo/env というファイルがあるので、そちらを source してあげるとよろしいかと思います。ひとまず、 cargo というコマンドが使えるようになっていたら大丈夫です。なお、「どうしても deb で」という方は cargo パッケージをインストールしてください。その際、rustc のバージョンが 2018 Edition である 1.31 以降である事が望ましいです。

3.2.2 Hello World してみる

では、cargo が使えるようになったところで、Hello World してみましょう。Rust のプロジェクトは、 cargo new で作成することができます。昔は--bin をつけないとライブラリ用のプロジェクトになってましたが、最近の cargo であればデフォルトがバイナリ用のプロジェクトになります。それでは実行してみましょう。

```
% cargo new hello
     Created binary (application) `hello` package
% tree
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

実行すると、こんな感じに Cargo.toml というファイルと、main.rs というソースファイルが作成されます。そう、Rust のソースファイルの拡張子は.rs です。そのため、Rust のプロジェクトのウェブサイトは、.rs ドメインで作られてる事がおおいです。ちなみに、.rs はセルビアのドメインです。

実は、この時点で Hello World の半分が完了しています。src/main.rs の中身を見てみましょう。

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

なんと、既に Hello World のコードが生成されています。なお、詳細は省きますが、 println!() は関数に見えますが、ビックリマークが付いている物は、Rust ではマクロであったりします。まぁ、深く付き会う前は、特に気にしなくて良いかと思います。ビルドして実行するには cargo run すれば OK です。

```
% cargo run
   Compiling hello v0.1.0 (/path/to/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/hello`
Hello, world!
```

3.3 Rust の構文等の紹介

それでは、ここからは簡単に Rust の構文等の紹介をしていきたいと思います。なお、私も勉強中の身ですので、微妙に嘘を書いていたらすいません。正確な事は、The Book[*6]と呼ばれる気合の入ったドキュメントがありますのでそちらを参考にしていただけたらと思います。

なお、Rust は比較的学習が難しいとか言われてますが、なんで難しいのかについて、個人的にはオライリーの Rust 本で引用されてる Quora に書かれたコメント[*7]がしっくり来たので引用しておきます。

> I've found that Rust has forced me to learn many of the things that I was slowly learning as "good practise" in C/C++ before I could even compile my code.

[*6] https://doc.rust-lang.org/book/

[*7] https://www.quora.com/What-do-C-C++-systems-programmers-think-of-Rust/answer/Mitchell-Nordine

私は最近会社の仕事の関係で C++ の勉強もするハメになってるのですが、まさに上記と同じ感想です。なにか、C++ がこなれて来て、C++11 あたりでようやく良い感じになった機能だけを取り入れていったのが Rust なんじゃないかなぁと。

3.3.1 変数宣言

変数の宣言は `let` キーワードを使います。型は、最近の言語っぽくコロンの後に書きますが、型が推論できるなら省略可能です。また、シャドーイングすることも可能で、指定がなければ immutable な変数になるので、mutable な変数を作る場合は `mut` キーワードを使います。具体的な例を以下に示します。

```
fn main() {
    let x = 1;
    println!("x = {}", x);
    let x = 1.25;
    println!("x = {}", x);
    x = 1;   // エラーになる（そもそも immutable だけど): expected floating-point number, found integer
    x = 1.5; // エラーになる: cannot assign twice to immutable variable

    let mut y: u32 = 1;
    y -= 1;
    y -= 1;
    println!("y = {}", y);
}
```

例の 2 つ目の `y -= 1;` ですが、デバッグビルドだとオーバーフローを検知して実行時にパニックを起こします。リリースビルド (`--release` オプション付きで `cargo` を実行) だとパニックは発生しませんが、もちろんおかしな値が表示されます。

なお、基本的な型は以下のようなものがあります。

整数 符号無し: `u8`, `u16`, `u32`, `u64`, `usize`
符号付き: `i8`, `i16`, `i32`, `i64`, `isize`
浮動小数点 単精度: `f32`、**倍精度:** `f64`
ブール `bool`
文字 `char` (Unicode の 1 文字)
文字列 `String`
- ただし、文字列リテラルの `str` があってとてもややこしい

配列 `[T; N]` (`T`: 型, `N`: 要素数)
ベクター `Vec<T>` (`T`: 型)
スライス `&[T]` (`T`: 型)

`String` と `str` が少々ややこしいですが、イメージとしては C++ の `std::string` と `char` の関係に近いような感じです。`String` の方は可変長で、メモリ上のイメージを図にすると図 3.3.1 のような感じです。`&`がついてる `str` は、この後述する参照になってます。ひとまず、以下がちょっとしたサンプルコードです。

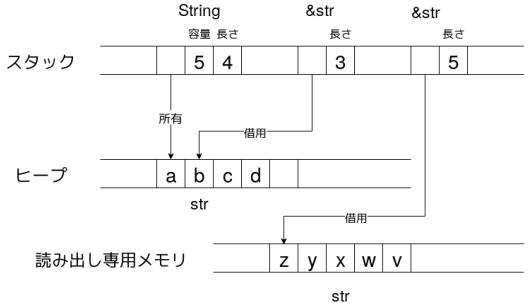


図 1　Rust における String と str の違い

```
fn main() {
    let a = "abcd".to_string(); // String::from("abcd"); でも可
    let b = &a[1..];
    let c = "zyxwv";

    let mut a = "abcd".to_string();
    //  ^^^  a を mutable な変数として宣言
    a.push_str("efg");  // OK

    let mut c = "zyxwv";

    // エラー!: no method named ‘push_str‘ found for type ‘&str‘ in the current scope
    c.push_str("ut");  // str には追加できないので push_str メソッドはない
}
```

3.3.2 関数定義

では、関数を定義してみましょう。Rust での関数定義は、fn キーワードと ‘->’ トークンを使います。まずは、単純に足し算を行う関数 add を定義してみます。

```
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}

fn main() {
    println!("{}", add(1, 2));
}
```

これを cargo で実行すると、(驚くような事は一切ないですが) 以下のような結果になります。

```
% cargo run
Compiling function v0.1.0 (/path/to/examples/function)
Finished dev [unoptimized + debuginfo] target(s) in 0.20s
Running ‘target/debug/function‘
3
```

なお、ここでは return を使いましたが、Rust ではセミコロンを抜かすと値を作るため、セミコロンを抜かして書いた以下のような add 関数も上記で定義した関数と同じ動作をします (セミコロンを書くとエラーになります)。

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

3.3.3 所有権

さて、ここから一気に Rustっぽい話になります。Rust では、メモリの安全性が考えられていると書きましたが、そのために所有権 (英語では ownership) という概念を使っています。C++ でもある概念ですが、たとえば以下のようなプログラムがあったとします。

```
fn main() {
    let i0 = 1;
    let i1 = i0;
    let i2 = i0;

    let s0: String = "hoge".to_string();
    let s1 = s0;
    let s2 = s0;
}
```

さて、どうなると思うでしょうか? 正解は、以下のように s2 に s0 を代入しているところでエラーになります。

```
error[E0382]: use of moved value: ‘s0‘
 --> ownership/src/main.rs:8:15
  |
6 |     let s0: String = "hoge".to_string();
  |         -- move occurs because ‘s0‘ has type ‘std::string::String‘, which does not implement the ‘Copy‘ trait
7 |     let s1 = s0;
  |              -- value moved here
8 |     let s2 = s0;
  |              ^^ value used here after move
```

Rust では、代入、関数の引数、関数の戻り値などで値の所有権が移動するため、一度所有権を移譲してしまったら、移譲元の変数はその後アクセスできなくなります (コンパイル時に怒られる)。ところが、上記のコードでは整数の方はエラーになりません。これは何故かというと、整数など一部の型 (サイズが固定で小さいものが主?) は移動せ

ずにコピーになります。実はエラーメッセージにある Copy trait というのがミソで、整数型はこの Copy trait を実装しているからそういった動作になっています。trait については後述します。

関数の引数でも所有権は移動するので、以下のようなコードもアウトです。

```
fn consume(_s: String) {}

fn main() {
    let h: String = "Hello World".to_string();
    consume(h);
    let hh = h;
}
```

ビルドすると、関数 consume コール時に h が移動してしまっているため以下のようなエラーになります。

```
12 |     let h: String = "Hello World".to_string();
   |         - move occurs because ‘h‘ has type ‘std::string::String‘, which does not implement the ‘Copy‘ trait
13 |     consume(h);
   |             - value moved here
14 |     let hh = h;
   |              ^ value used here after move
```

逆に、関数の戻り値でも所有権を移動できるので、以下のようなコードが書けたりします。

```
fn generate(i: i32) -> Vec<i32> {
    let mut v = Vec::new();
    v.push(i);
    return v;
}

fn main() {
    let v1 = generate(10);
    let mut v2 = generate(100);
    v2.push(101);
    println!("v1: {:?}", v1); // v1: [10]
    println!("v2: {:?}", v2); // v2: [100, 101]
}
```

3.3.4 参照と借用

関数呼び出しなどでは、所有権を渡さずに値を使いたい場合が多々あるかと思いますが、その場合には参照を使います。なお、Rust では借用とも言います。記法としては、C++ と同じく&を使うのですが、C++ とは、

- 借用する側 (参照される側) に ‘&’ を付ける
- 関数の呼び出し元と呼び出し先両方に ‘&’ が必要

といった点で異なる感じになります。先程、所有権の移動でエラーになった関数呼び出しのコードを参照をつかってエラーにならないように書き換えたものが以下になります。

```
fn consume(_s: &String) {}

fn main() {
    let s0: String = "hoge".to_string();
    let s1 = &s0;
    let s2 = &s0;

    let h: String = "Hello World".to_string();
    consume(&h);
    let hh = h;
}
```

このコードであれば、関数 consume をコールしても、h の所有権はうつっていないため、その後 hh に所有権を移譲することができます。参照の解決には、以下のように*を使います。

```
fn main() {
    let i0 = 5;
    let i1 = &i0;

    println!("i0: {}, i1: {}", i0, *i1);
}
```

が、関数で使う場合だったり、色々な場面で省略できたりするみたいです。

```
fn catlength(a: &String, b: &String) -> usize {
    return a.len() + b.len();
}

fn main() {
    let s0 = "hoge".to_string();
    let s1 = "fuga".to_string();
    println!("{}", catlength(&s0, &s1));
}
```

3.3.5 ライフタイム

Rust には、GC はなくて C 言語のように変数スコープがあり、スコープを外れた変数は随時消えていきます。例としては、以下のようなコードを書くと、ドロップされた後に値を使おうとしているためコンパイル時の怒られます。

```
fn main() {
    {
        let a = 0;
    } // ここでドロップ
    println!("{}", a);
}
```

実行すると以下のように怒られます。

```
error[E0425]: cannot find value ‘a‘ in this scope
--> lifetime/src/main.rs:5:20
  |
5 |     println!("{}", a);
  |                    ^ not found in this scope
```

参照している場合とて怒られます。例として、以下のようなコードを書きます。

```
fn main() {
    let r;
    {
        let b = 0;
        r = &b;
    }
    println!("{}", r);
}
```

コンパイル時に怒られます。

```
error[E0597]: ‘b‘ does not live long enough
--> lifetime/src/main.rs:11:9
   |
11 |         r = &b;
   |         ^^^^^^ borrowed value does not live long enough
12 |     }
   |     - ‘b‘ dropped here while still borrowed
13 |     println!("{}", r);
   |                    - borrow later used here
```

でも、実はこの**「コンパイル時の怒られる」**というのがミソなのです。C/C++ でよくやる解放済みのメモリアクセスとかがコンパイル時にみつかるのです。素敵!! とはいえ、ややこしいのが関数です。以下がエラーになります。

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

こんな感じで怒られます。

```
error[E0106]: missing lifetime specifier
--> lifetime/src/main.rs:1:33
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |                                 ^ expected lifetime parameter
  |
= help: this function’s return type contains a borrowed value, but the signature does not say whether it is \
borrowed from ‘x‘ or ‘y‘
```

このコードの問題はなにかというと、実行するまで x と y(どちらも参照) のどちらを返すかわからないので、戻り値のライフタイムがどうなるかわからないので、その後の処理がチェックできないという事で怒られています。では

どうするかというと、コンパイラが言っている通り lifetime parameter を追加します。具体的には、生存期間'a という表現を用いて以下のように書きます。

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

これによって、この関数 longest の戻り値は、x か y の短い方のライフタイム以上のライフタイムを持つという記載になり、コンパイラはこれによって値の借用のチェックを行ないます。

3.3.6 構造体

Rust には class がなく、構造体 struct を使います。定義は C 言語の構造体と同じような (型は後置になります) 感じですが、宣言する場合は各要素の名前を逐一書く必要があります。ただし、要素名と同名の変数で初期化するときのみ省略可能です。あとは、他の変数から値をコピーして一部だけ値を入れるといった記法もあります。

```
struct Point {
    x: f64,
    y: f64,
}

fn main() {
    let _p0 = Point { x: 1.0, y: 1.0 };

    let x = 0.0;
    let y = 0.0;
    let _p1 = Point { x, y }; // 変数名とメンバー名が一緒
}
```

なお、構造体ですが、メソッドを生やすことができます。その際、impl キーワードを用いて、構造体の定義から離れた場所に書きます。各関数の最初の引数は特別な引数である self への参照である必要があります。メンバーの値を書き換えたい場合は mut を付けます。

```
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn abs(&self) -> f64 {
        (self.x * self.x + self.y * self.y).sqrt()
    }
}

fn main() {
    let p0 = Point { x: 1.0, y: 1.0 };
    println!("{}", p0.abs()); // => 1.4142135623730951
}
```

3.3.7 ジェネリック

C++ のテンプレートや Java のジェネリクスのように、Rust でも型だけ異なるような関数や構造体を一般的な形で記述することができます。

```
struct GenPoint<T> {
    x: T,
    y: T,
}

fn main() {
    let _p1 = GenPoint::<i32> { x: 1, y: 1 };
    let _p2 = GenPoint { x: 1, y: 1 };
    let _p3 = GenPoint { x: 1.5, y: 2.0 };
}
```

3.3.8 trait

Java や Go のインターフェース的なものとして、Rust にはトレイト (trait) というものがあります。さきほど出てきた Copy trait もこれになります。トレイトは trait キーワードで定義しますが、たとえばダックタイピングな

trait を書くと以下のようになります。

```
// Duck は……
trait Duck {
    fn walk(&self);   // walk して
    fn quack(&self);  // quack するもの!!
}
```

この状態で、この Duck trait を実装する構造体を impl キーワードで以下のように作成します。

```
struct RealDuck {}

impl Duck for RealDuck {
    fn walk(&self) {
        println!("duck walking");
    }

    fn quack(&self) {
        println!("quack")
    }
}

struct Dog {}

impl Duck for Dog {
    fn walk(&self) {
        println!("dog walking");
    }

    fn quack(&self) {
        println!("bow")
    }
}
```

そうすると、以下のように Duck 型を引数とする関数 test_duck() に RealDuck と Dog のどちらも渡すことができるようになります。

```
fn test_duck(duck: &Duck) {
    duck.walk();
    duck.quack();
}

fn main() {
    let duck = RealDuck {};
    let dog = Dog {};

    test_duck(&duck);
    test_duck(&dog);
}
```

なお、引数に複数の trait を実装した型を要求することができます。たとえば、

```
fn notify(item: impl Summary + Display) {
```

は 2 つの trait、Summary と Display を同時に実装した型を引数 item に要求します。これだと読みづらい感じもするので、以下の書き型もできます。

```
fn notify<T: Summary + Display>(item: T) {
```

また、引数が一杯になってきたら上記でも辛いので、以下のような where をつかって記法もあります。

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

3.3.9 enum

さて、enum です。個人的にはかなり気にいってますが、Rust の enum は、ちょっとおかしなくらいに色々とできます。もちろん、C 言語のような以下のような enum は作れます。

```
enum Fruits {
    Apple,
    Banana,
    Orange,
    Peach,
}

fn main() {
    let _hoge = Fruits::Apple;
}
```

が、Rust の enum は、各値の中にさらに値を持たせることができます。しかも、型や値の個数はバラバラで構いません。たとえば、以下のような enum を定義できます。なお、ここで出てきている (u8, u8, u8, u8) というのは 8-bit 符号無し整数を 4 つ持つタプル型です。

```
enum IpAddr {
    V4((u8, u8, u8, u8)),
    V6(String),
}
```

このように定義してあげると、実際の値を作る際、以下のように値を入れて enum の値を作ることができます。

```
fn main() {
    let v4_addr = IpAddr::V4((127, 0, 0, 1));
    let v6_addr = IpAddr::V6("::1".into());
}
```

このようにして作った値は、match を使って値を取り出せます。C 言語の switch 文的な感じで使えます。以下のコードは enum 値全てのケースを記述していますが、default 的な処理は "_ => { /* **処理** */ }" として書けます。

```
fn extract_addr(addr: &IpAddr) {
    match addr {
        IpAddr::V4(a) => {
            println!("{}.{}.{}.{}", a.0, a.1, a.2, a.3);
        }
        IpAddr::V6(a) => {
            println!("{}", a);
        }
    }
}

fn main() {
    extract_addr(&v4_addr); // => 127.0.0.1
    extract_addr(&v6_addr); // => ::1
}
```

3.3.10 Option<T>と Result<T,E>

Rust には例外も nullptr もありませんが、その代わりにジェネリックな enum である Option<T>と Result<T,E>を使ってエラー等を処理します。それぞれの機能をざっくり説明すると、

- Option<T>
  - 値がない可能性がある事を示す (nullptr の代わり)
- Result<T,E>
  - 失敗する可能性がある事を示す (例外の代わり)

といった感じでしょうか。それぞれの定義は以下のようにされています。

```
pub enum Option<T> {
    None,
    Some(T),
}

pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

例えば、1 つ目のコマンドライン引数を受けとり、それを i32 型に変更して表示する関数を書くと次のようになります。まず引数が一つも無い場合があるため、引数があれば Some() の中身に引数が、なければ None が取り出され

ます。その後、Some() の中身を取り出しますが、i32 型としてパースできない場合があるため、成功したか失敗したかを Ok() か Err() かで判定しています。

```
use std::env;

fn main() {
    let i = match env::args().nth(1) {
        None => {
            panic!("no arguments");
        }
        Some(arg) => match arg.parse::<i32>() {
            Ok(fig) => fig,
            Err(e) => {
                panic!("error: {}", e);
            }
        },
    };

    println!("i: {}", i);
}
```

実際には、上記の処理をもうちょっと綺麗に書けるように?演算子や expect() メソッドなどが定義されています。が、ここでは詳細は省きます。

3.3.11 crate

Rust のプログラムは、クレート (crate) と呼ばれる単位の組み合わせで構成されます。今回は、詳細については割愛して、crates.io[*8]にて公開されているクレートの使い方について簡単に紹介します。といっても、実はとても簡単で、cargo の設定ファイルの Cargo.toml に crates.io で使いたいクレートのページに掲載されている式を貼り付けるだけです。

今回はオプション解析のクレートである clap[*9]を使ってみましょう。サイトにある式を、Cargo.toml の [dependencies] の項目に追記してあげれば OK です。

```
[dependencies]
clap = "2.32.0"
```

では、これをつかって、簡単なプログラムを作ってみます。とりあえず、-n オプションだけあるような似非 echo コマンドを以下のように書いてみました。

```
use clap::{App, Arg};

fn main() {
    let m = App::new("echo")  // コマンド名
        .arg(Arg::with_name("STRING").multiple(true))  // 複数の引数
        .arg( // '-n' を定義
            Arg::with_name("n")
                .short("n")
                .help("do not output the trailing newline"),
        )
        .get_matches();  // 実行

    let out = match m.values_of("STRING") {
        //          ↓地味にイテレータを使用
        Some(v) => v.collect::<Vec<&str>>().join(" "),
        None => "".into(),
    };

    print!("{}", out);

    if !m.is_present("n") {
        println!("");  // '-n' オプションがなければ改行
    }
}
```

Rust 2018 Edition と呼ばれる 2018 年の年末にリリースされたバージョン以降では、これまで必要だった extern crate clap; という記述が不要になっています。

ということで、ここまでかなり高速に Rust の構文等の紹介をしてきましたが、まったくもって極一部ですので、興味があれば The Book を読んでもらえればと思います。

[*8] https://crates.io/

[*9] https://clap.rs/

3.4 Debian パッケージにしてみる

ここでようやく Debian な話題ですが、ひとまずここでは先程作成した clap で作った似非 echo コマンドを debian パッケージにしてみるということをやります。

Rust の単一バイナリの debian パッケージ化について、一番簡単なのは `cargo` のサブコマンド用の `cargo-deb`[*10] を用いる方法があるようです。あるようなんですが……残念ながらこのツール、とても公式に入れられないような deb しか吐かないようです。ドキュメントもないし云々。まぁ、/usr/bin にえいやと入れたいならアリかもですが。

で、さすがにそれだとアレなので、もうちょっとちゃんとした deb を作るべく、Debian Rust packaging team の Wiki[*11]に、それなりにしたがってみようかと思います。どうやら、crates.io にある crate については、debcargo[*12] というツールを使うと良い感じに処理してくれるようです。が、今回は crate.io にはアップされていないもので作りたいという話なんですが、debcargo は対応していなさそうです。そのため、debcargo が使っている dh-cargo をつかって地道に対応してみることにしました。

まずは、dh_make を実行して debian ディレクトリを作り、できあがった debian/rules の build ルールの dh にオプション`--byildsystem cargo` を追加することと、debian/control の Build-Depends に dh-cargo と librust-clap-dev を追加してみました。ひとまずここでビルドしてみます。

```
% debuild-pbuilder
-> Attempting to satisfy build-dependencies
-> Creating pbuilder-satisfydepends-dummy package
Package: pbuilder-satisfydepends-dummy
Version: 0.invalid.0

... (snip) ...

dh_autoreconf -O--buildsystem=cargo
dh_auto_configure -O--buildsystem=cargo
cp: './debian/cargo-checksum.json' を stat できません: そのようなファイルやディレクトリはありません
```

なにかファイルがないと言われてしまいました。Wiki を見ると、どうやら upstream の crate のチェックサムを含めたファイルを用意してやる必要があるようです。これは、/usr/share/cargo/registry 以下に deb でインストールされたライブラリたちを使うためのようなのですが、あまり Wiki を読んでも作り方がわかりませんでした。どうも、cargo-vendor というツールを利用していそうなのですが……。とか思っていたのですが、

```
% apt source ripgrep
% cat rust-ripgrep-0.10.0/debian/cargo-checksum.json
{"package":"Could not get crate checksum","files":{}}
```

と、公式の ripgrep の deb のソースをみると、このファイルが哀しいことになっていました。それでもちゃんとビルドは通るようです。ためしに、debian/cargo-checksum.json を touch で空ファイルとして作ってみたら、普通にビルドが進んでしまいました。いったんわすれます……。

ところが、まだコケます。

```
debian cargo wrapper: running subprocess (['env', 'RUST_BACKTRACE=1', '/usr/bin/cargo', '-Zavoid-dev-deps', \
'build', '--verbose', '--verbose', '-j4', '--target', 'x86_64-unknown-linux-gnu'],) {}
error: failed to select a version for the requirement `textwrap = "= 0.10.0"`
candidate versions found which didn't match: 0.11.0
location searched: directory source `/path/to/clap-test/debian/cargo_registry` (which is replacing registry \
`https://github.com/rust-lang/crates.io-index`)
required by package `clap v2.32.0`
```

clap が依存している textwrap のバージョンが 0.10.0 だけど、手元にあるのは 0.11.0 だと。たしかに、textwrap の deb のバージョンは 0.11.0 でした。が、そうなるとおなじく clap を使っている ripgrep のビルドが通っているのが解せません。と調べていくと、GitHub のリポジトリ[*13]に置いてあるバージョンと ripgrep の deb の orig ソー

[*10] https://github.com/mmstick/cargo-deb

[*11] https://wiki.debian.org/Teams/RustPackaging

[*12] https://salsa.debian.org/rust-team/debcargo

[*13] https://github.com/BurntSushi/ripgrep

スを見比べると、Cargo.toml が若干異なり、Cargo.lock が編集されていることがわかりました。そして、消された Cargo.lock の中には、textwrap の 0.10.0 のチェックサムが記載されています。そもそも、Cargo.toml はで開発者がざっくりとした依存を書くファイルで、Cargo.lock は、cargo が実情に基づいて自動で生成し、`cargo update` などすることによってバージョンが更新されるものです。

で、確認していくと、どうやら/usr/share/cargo/registry 以下にある textwrap の Cargo.toml の textwrap の依存が crates.io 登録時に以下のように書きかわっていることがわかりました。

```
% grep -A 1 dependencies.textwrap debian/cargo_registry/clap-2.32.0/Cargo.toml
[dependencies.textwrap]
version = ">= 0.10, < 0.12"
```

ちなみに、GitHub の clap のコードには以下としか書いてないです。

```
[dependencies]
bitflags               = "1.0"
unicode-width          = "0.1.4"
textwrap               = "0.10.0"
```

ということで、変更の基準はまったくわかってませんが (今後の課題とさせていただきます!!)、今回の似非 echo コマンドについては、Cargo.lock を消せば最後までビルドされました。lintian のエラーも警告も取れてないですが、ひとまず動きますよ!!

```
% dpkg -I rust-clap-test_0.1.0-1_amd64.deb
new Debian package, version 2.0.
size 258160 bytes: control archive=648 bytes.
360 バイト、    11 行      control
285 バイト、     4 行      md5sums
Package: rust-clap-test
Version: 0.1.0-1
Architecture: amd64
Maintainer: Katsuki Kobayashi <rare@tirasweel.org>
Installed-Size: 770
Depends: libc6 (>= 2.18), libgcc1 (>= 1:4.2)
Section: unknown
Priority: optional
Homepage: <insert the upstream URL, if relevant>
Description: <insert up to 60 chars description>
<insert long description, indented with spaces>

% dpkg -c rust-clap-test_0.1.0-1_amd64.deb
drwxr-xr-x root/root         0 2019-03-23 18:06 ./
drwxr-xr-x root/root         0 2019-03-23 18:06 ./usr/
drwxr-xr-x root/root         0 2019-03-23 18:06 ./usr/bin/
-rwxr-xr-x root/root    776600 2019-03-23 18:06 ./usr/bin/clap-test
drwxr-xr-x root/root         0 2019-03-23 18:06 ./usr/share/
drwxr-xr-x root/root         0 2019-03-23 18:06 ./usr/share/doc/
drwxr-xr-x root/root         0 2019-03-23 18:06 ./usr/share/doc/rust-clap-test/
-rw-r--r-- root/root       197 2019-03-23 18:06 ./usr/share/doc/rust-clap-test/README.Debian
-rw-r--r-- root/root       190 2019-03-23 18:06 ./usr/share/doc/rust-clap-test/changelog.Debian.gz
-rw-r--r-- root/root      1693 2019-03-23 18:06 ./usr/share/doc/rust-clap-test/copyright
```

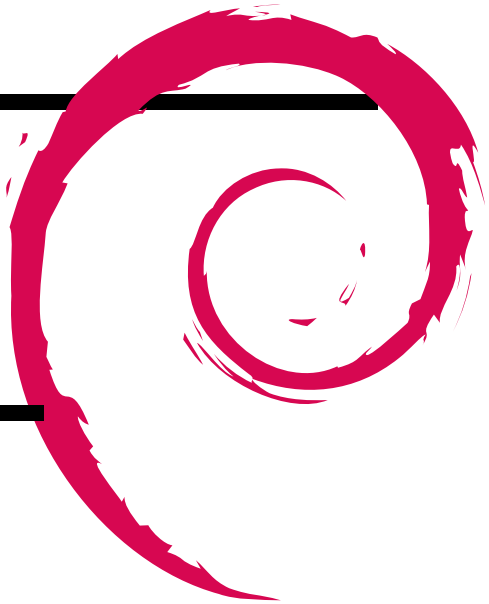
3.5 まとめ

ということで、しりすぼみ感は拭えませんが、結局 Rust な deb を作るには、現状では crates.io に登録して debcargo をベースにつくるというのが公式のようでした。が、もちろん dh-cargo があるので crates.io に登録しなくても deb にはできそうです。cargo 自体は、外部クレートに git リポジトリを指定したりできるので、debcargo も git リポジトリに対応できたら良いですが、今回の Cargo.toml/Cargo.lock を crates.io が編集している兼ね合いもあるので、色々と大変かもしれません。

というか、そもそも librust-*パッケージ達の依存関係はあやしいのかもしれません。つい先日も、ripgrep がビルドできないというバグも報告されていましたし[*14]。とは言え、debcargo 自体は粛々と開発が進んでいるようで、これを書いてる直前くらいにも、buster には入らないけど post-install で test ができるようになったり[*15]しているようです。なんにせよ、Rust 自体は良い言語かと思うので、今後流行っていけば良いなぁと思います。

[*14] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=920958

[*15] https://alioth-lists.debian.net/pipermail/pkg-rust-maintainers/2019-March/005296.html



4 開催の予定

関西 Debian 勉強会

Debian 勉強会資料

2019 年 3 月 24 日　　初版第 1 刷発行
関西 Debian 勉強会　（編集・印刷・発行）