Rust 入門

Rust とは

速い

- C++と同じくらい速いらしい。ガベージコレクションがない

安全

- 所有権の概念のおかげで（ガベージコレクションがなくても）メモリ安全、スレッド安全

生産性が高い（らしい）
マルチパラダイム

- 手続き的にも関数型的にも書ける

そしてオープンソース

インストール

- Windows : Windows で Rust 用の開発環境を設定する
  - Windows では Microsoft C++ Build Tools が必要
  - WSLなら `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
- Mac : macOSでRustのローカル開発環境を整えるための手順2022
  - WSLと同じコマンドでよさそう
- `cargo --version` でバージョンが表示されればOK
- 適宜 VSCode の拡張機能を入れる（rust-analyzer, CodeLLDB）
- `rustup self update`, `rustup update` でアップデート

インストールしなくてもRust Playgroundでコードを実行できる

VSCodeでのデバッグ、実行

- `cargo new {project name}`でプロジェクトが作成される（あるいは`cargo init`でそのフォルダをプロジェクト化する）
- **プロジェクトフォルダを**VSCodeで開き、ダイアログで`OK`, `Yes`を押すとlaunch.jsonが作られる（参考）
- 次から`F5`でデバッグ、`Ctrl+F5`で実行できるようになる
- `rustc {filename}.rs`でコンパイルされる
- `cargo build`でビルド。`--release`オプションを付けられる
- `cargo run`で（必要ならビルド+）実行

Rust の仕様

Rustでは所有権を通してオブジェクトに触る

- スコープを抜ける or 所有権を渡すともとの変数が持っていた所有権は消失する。**参照**を渡すとよい
- 所有権がすべて失われるとオブジェクトが破棄される

	不変	可変
原本	`let a = 10;`	`let mut b = 20;`
参照	`let aref = &a;`	`let bref = &mut b;`

- 関数呼び出し時に**借用チェック**がクリアされる必要がある
  - 不変参照と可変参照が同時に存在しない
  - 可変参照は1つしか存在しない

```
fn main() {
    let mut a = 10;           // mutable object
    let a_ref1 = &a;          // reference
    let a_mut_ref1 = &mut a;  // mutable reference
    // let a_mut_ref2 = &mut a;  // この時点で a_ref1, a_mut_ref1 は存在しない
    // let a_ref2 = &a;          // この時点で a_mut_ref2 は存在しない
    *a_mut_ref1 = 30;
    a = 20;
    println!("{}", a);        // borrow check!! - Error!
    //println!("{} {}", a_ref1, a_ref2); // borrow check!! - Error!
    // println!("{}", a_ref2);   // borrow check!! - OK
}
```

コピートレイト

- コピートレイトの実装された型のコピーでは、所有権は移動せずオブジェクトがコピーされる
- Rustのプリミティブ型、不変参照はコピートレイトを実装している
- 可変参照はコピーされない
- オブジェクト指向言語でプリミティブ型はdeep copyされるのに似ている
- **トレイト境界**：あるトレイトを実装しているか否かで型を制限できる

```
fn copy_trait_check<T:Copy>(_:T){}
```

に渡せる型はコピートレイトを実装している

データ型

- 整数、`f32`,`f64`,`bool`,`char`,タプル、配列など
- 数字リテラルは`_`で区切れる
- `0x`,`0o`,`0b`が16,8,2進数リテラルの接頭辞

もろもろ

- パターン分解できる：`let (x,y,z) = (1,2,3);`。無視するときは`_`
- 型を明示的に指定できる（普段はrust-analyzerが推定される型を表示している）
- 同じスコープ内で変数を使い回せる
- シャドーイングが有効
- キャストは`as`を使う：`let c = 13u8 as u32 + 7u32;`

配列・スライス

- 配列：`let a = [1,2,3,4,5];`
- スライス：`let s = &a[1..3];`
- 範囲の指定 `x..y` ：$x \leq i < y$, `x..=y` ：$x \leq i \leq y$, `..y` ：$i < y$ など
- 範囲指定は `for` 文でも使える

```
let mut sum = 0;
for i in 1..10 {
  sum += i;
}
```

- 文字列リテラル、文字列のスライスは `&str` 型
- `String::from("uouo")` は `String` 型

関数

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

- 引数は（返り値がある場合は返り値も）型を指定する必要がある
- 引数でもパターン分解を使える

```
fn add(&(x, y): &(i32, i32)) -> i32 {
  x + y
}
fn main() {
  println!("{}", add(&(1, 2)));
}
```

制御構文

- `if 条件 { ... } else if 条件 { ... } else { ... }`
- `if`文も式を返す：`let mod2 = if a % 2 == 1 {"odd"} else {"even"};`
- `loop`文は`break`で値を返せる（条件式はない）

```
fn main() {
  let mut a = 7; let mut ret = 1;
  let a0 = a;
  let result = loop {
    ret *= a; a -= 1;
    if a == 0 { break ret; }
  };
  println!("{}! = {}", a0, result);
}
```

- `for`,`while`は値を返さない（`()`を返す）

構造体

```
struct Cmplx { real: f32, imag: f32 }
fn main() {
  let mut z = Cmplx { real: 0.5, imag: 0.9 };
  z.imag = -3.;
  fn build(real: f32, imag: f32) -> Cmplx { Cmplx {imag, real} }
  let w = Cmplx {imag: 4., ..build(1., 2.)};
  println!("w = {} + {} i", w.real, w.imag); // w = 1 + 4 i
  println!("z = {} + {} i", z.real, z.imag); // z = 0.5 + -3 i
}
```

- `let mut` するとフィールド全てが可変になる
- 変数名と一致しているフィールド名は省略できる
- 指定しなかったフィールドは `..` 後のオブジェクトのフィールドを束縛する

タプル構造体

```
struct Cmplx(f32, f32);
fn main() {
  let z = Cmplx(1., 2.);
  let Cmplx(x, y) = z;
  println!("{} + {} i", z.0, z.1);
  println!("{} + {} i", x, y);
}
```

- `.0`などでアクセスする
- 要素の分解もできる

メソッド、関連関数

```
struct Cmplx { real: f32, imag: f32 }
impl Cmplx {
  fn abs(&self) -> f32 { self.real * self.real + self.imag * self.imag }
  fn new(real: f32, imag: f32) -> Self { Self { real, imag } }
}
fn main() {
  let z = Cmplx::new(3., 5.);
  println!("{}", z.abs());
}
```

- 書き換えるときは`&self`の代わりに`&mut self`
- `self`だと呼び出し元のオブジェクトの所有権がメソッドに渡り、元の場所でそのオブジェクトを使えなくなる
- `Self`は型エイリアス

列挙型

```
enum OpticalDisc { CD(u32), BD(String, u32) }
fn main() {
  let bd = OpticalDisc::BD(String::from("ROM"), 120);
}
```

- 列挙子に値を持たせられる。持たせる値の型は列挙子ごとに違っていていい
- メソッドも定義できるしトレイトも実装できる
- 列挙子に持たせたオブジェクトは`match`文で取り出せる

デフォルトで用意される列挙型の例

```
enum Option<T> = { Some(T), None }
enum Result<T, E> = { Ok(T), Err(E) }
```

ジェネリクス

```
use std::f32::consts::PI;
enum OpticalDisc<T> { CD(T), BD(String, T) }
impl<T> OpticalDisc<T> {
  fn newBD(r:T) -> OpticalDisc<T> {
    OpticalDisc::BD(String::from(""), r)
} }
impl OpticalDisc<f32> {
  fn area(&self) -> f32 {
    match *self {
      OpticalDisc::CD(r) => r.powi(2) * PI / 4.,
      OpticalDisc::BD(_, r) => r.powi(2) * PI / 4.
} } }
fn main() {
  let bd = OpticalDisc::<f32>::newBD(120.);
  println!("area of bd is {}", bd.area());
}
```

- ジェネリック型：任意の型を受け取れるようにしてコードの重複を回避する仕組み
- ジェネリック型のメソッドを定義できる。特定の型のときのメソッドを定義することもできる
- コンパイル時は型が分かるので型に特化したコードが生成される（**単相化**）
- **turbofish演算子** `::<...>` で型を明示的に指定できる

パターンマッチ

```
fn main() {
  let a: Option<String> = Some(String::from("guee"));
  match a {
      Some(ref x) => println!("{}", x),
      _ => {println!("uouo"); ()}
  }
  println!("{:?}", a);
}
```

- パターンマッチはマッチした腕だけが評価される（短絡評価）
- 残りのパターンは`_`で一括して受け取れる
- パターンマッチで所有権が移るので、参照でマッチさせるのがよい。
- `_`以外が1つなら`if let Some (ref x) = a {println!("{}", x)}`のように短く書ける
- **マッチガード**：`識別子 条件 =>`のように条件を追加できる

自然数？

```
enum Nat { S(Box<Nat>), O }
fn main() {
  let one = Nat::S(Box::new(Nat::O));
}
```

エラー処理、コンビネータ

- `panic!("メッセージ")`でメッセージとともに強制終了できる
- `option.unwrap()`は内部で`match option { Some(v) => v, None=>panic!(...) }`のような処理をしてくれる
  - `Result`型についても使える
  - `expcept`ならメッセージも追加できる
- `?`演算子を`Option`,`Result`型に作用させて、値がないときに呼び出し元に返す処理を実現できる。
- `let ret = open()?.read()?.replace()?.write()?.close()?;`みたく書ける（**コンビネータ**）
- `map`,`and_then`などを活用して簡潔に書ける

トレイト（インターフェースみたいなもの）

```
struct Cmplx { real: f32, imag: f32 }
pub trait Distance {
  fn abs(&self) -> f32;
  fn name(&self) -> &str { return "Distance" }
}
impl Distance for Cmplx {
  fn abs(&self) -> f32 {
    self.real.powi(2) + self.imag.powi(2)
  }
  fn name(&self) -> &str { return "Complex" }
}
fn main() {
  let z = Cmplx { real: 1., imag: 2. };
  println!("{} abs={}", z.name(), z.abs());
}
```

- 構造体にトレイトの実装を付与できる（構造体をトレイトのインスタンスにする）
- デフォルトメソッドをオーバーライドできる
- トレイトはトレイトを継承できる：`pub trait Comparable : Distance{ ... }`
- トレイト境界を指定できる：`fn comp<T: Distance>(d1: &T, d2: &T) { ... }`
  - `fn name:(d1: &impl Distance)`,
  - `fn comp(d1: &T, d2: &T) where T: Distance`のようにも書ける
  - `+`で複数のトレイトを指定できる
  - ジェネリック型にもトレイト境界を設定できる

derive属性、Fromトレイト

```
#[derive(Debug)]
struct Cmplx { real: f32, imag: f32 }
impl From<f32> for Cmplx {
  fn from(real: f32) -> Self {
    Cmplx { real, imag: 0. }
  }
}
fn main() {
  let z = Cmplx::from(1.);
  let w: Cmplx = (2.).into();
  println!("{:?} {:?}", z, w);
}
```

- 実装したいトレイトを型の定義時に指定できる
  - よく使われるトレイト：`Copy`, `Clone`, `Debug`, `(Partial)Eq`, `(Partial)Ord`
- `From`トレイトを実装すると`into`メソッドが使える

メモリについて

- リソースの確保、解放とオブジェクトの生成、破棄が同時
- `*`（参照外し：参照先のオブジェクトを操作する）は`Deref`トレイト（可変なら`DerefMut`）を実装すると実現できる
- `Drop`トレイトを実装して参照破棄時の挙動を設定できる
- メモリ領域は3種類ある
  - **データメモリ**：静的データを格納する
  - **スタックメモリ**：関数呼び出し時の引数などの一時的なデータを格納する
  - **ヒープメモリ**：`Box::new(...)`に渡されたオブジェクトが格納される
- **参照カウンタ**：原本の所有権が破棄されても仮の所有権を保持できる。`Rc::clone`に参照を渡せばよい。

内部可変性（`use std::cell::Cell;`）

- コピートレイトを実装した型を内部可変にするとき
  - `let a = Cell::new(10);`で作成、`a.get()`で取り出し、`a.set(20)`で代入
  - `a.replace(20)`はpopしてpush、`a.into_inner`で内部オブジェクトを取り出す
  - 不変・可変の検査が（コンパイル時ではなく）実行時に行われる（？）
- else
  - `RefCell`を使う
  - `.borrow()`で不変参照、`.borrow_mut()`で可変参照が返される
  - 可変・不変参照が満たすべき条件は（実行時に）満たしている必要がある

クロージャ（無名関数）

```
fn returns_closure() -> impl Fn(i32) -> i32 {
  |x| x + 1
}
fn main() {
  println!("{}", returns_closure()(2));
}
```

- クロージャは動的サイズ型なので`impl`を外すとまずい
- `impl`の代わりに`Box< ... >`を使ってもよい

コレクション

- ヒープメモリに置かれる

Vec

- `let v: Vec<i32> = Vec::new();` とか `let v = vec![1,2,3];`
- `for i in &v { ... }` で取り出せる

String

HashMap（連想配列）

```
use std::collections::HashMap;
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
let score = scores.get(&String::from("Blue"));
```

イテレータ

```
fn main() {
  let a1 = [1, 2, 3];
  let a2 = [4, 5, 6];
  let iter = a1.iter().zip(a2.iter().map(|x| x*x));
  for (i, j) in iter {
    println!("{}, {}", i, j); // 1, 16 \n 2, 25 \n 3, 36
  }
  let b = [0i32, 1, -3, 2, -4, 3, -1];
  let possum = b.iter().filter(|x| x.is_positive()).fold(0, |acc, x| acc + x);
  println!("{}", possum); // 6
}
```

- `collect`, `enumerate`, `for_each`（イテレータを返さない）, `inspect`（`map`の副作用許可版）
- `iter()`（`&self`を返す）の他に`iter_mut()`（`&mut self`）, `into_iter()`（`self`）などがある

さらに詳しく

- Tour of Rust：コードを実行しながら学べる。英語に抵抗がなければおすすめ。

参考文献

- Rust入門
- Rust入門（PDF）