

# Rust 入門

2021/2/1

すべてのページに、目次への  
リンクが隠れています

# 目次

- |     |                              |     |             |     |           |
|-----|------------------------------|-----|-------------|-----|-----------|
| 1.  | → Rust                       | 13. | → ジェネリクス    | 25. | → マクロ     |
| 2.  | → 最初に                        | 14. | → match 式   | 26. | → イテレータ   |
| 3.  | → Hello world!               | 15. | → エラー処理     | 27. | → コレクション  |
| 4.  | → 基本                         | 16. | → コンビネータ    | 28. | → Cargo   |
| 5.  | → 所有権                        | 17. | → トレイト      | 29. | → モジュール   |
| 6.  | → コピートレイト                    | 18. | → RAII      | 30. | → ユニットテスト |
| 7.  | → データ型                       | 19. | → クロージャ     | 31. | → Tips    |
| 8.  | → 関数                         | 20. | → 動的と静的     | 32. | → 最後に     |
| 9.  | → 制御式 (if, while, loop, for) | 21. | → ライフタイム    | 33. | → 参考      |
| 10. | → スライス                       | 22. | → 並列処理      |     |           |
| 11. | → 構造体                        | 23. | → 所有権まとめ    |     |           |
| 12. | → 列挙型                        | 24. | → derive 属性 |     |           |

# Rust

- **Rust** はマルチパラダイムプログラミング言語です。Rust は**静的型付け** (**statically-typed**) , **式ベース** (**expression-based**) であり, 手続き型・関数型プログラミングの両方を実装することができます。また, オブジェクト指向を言語としてサポートしている訳ではありませんが, オブジェクト指向プログラミングも制限付きで実装することができます。
- Rust はパフォーマンス, 信頼性, 生産性に重点を置き, システムプログラミング言語として適した言語を目指しています。

# Rust

- 個人的に Rust は関数型プログラミング言語である Haskell に大きく影響を受けており、ほとんどの部分が Haskell から引き継いでいるように思えます。ただし、純粋関数型でもなく、モナドというもの也没有せん。そこにポインタや参照といった別の言語の概念を取り入れ、さらに所有権といった独自の機能を組み込んだものです。これは、用語や機能が既存のプログラミング言語の概念に似てはいますが、必ずしも一致していないため、混乱しやすいところです。なので、最初は Rust を全く新しい言語として扱ったほうがいいのかもかもしれません。

# 最初に

- Rust には公式ドキュメント [The Rust Programming Language Book](#) があります。これはかなり丁寧で豊富な内容が含まれているわけですが、説明が冗長であり、量も多いので読むのに時間がかかります。そのため、手っ取り早く Rust を始める場合は [Tour of Rust](#) を利用した方がいいです。手軽にコードを動かしながら進められるのでとても解りやすいです。順番としては Tour of Rust をやり終えてから公式ドキュメントまたは他の参考サイトを参照するのが良いと思います。

# 概要

- これから Rust について解説していきます。注意として、わかり易さ・明確さを重視しているため、公式ドキュメントで使用している用語、およびその意味とは異なるところがある場合があります。また、読者対象には何かしらのプログラミング言語を学んでいる人を想定しています。
- Rust は簡単な言語ではありません。プログラミング初心者の場合は、まず別の言語から覚えることをオススメします。

# Hello world!

- 以下は、おなじみの Hello World プログラムです。非常に単純で直感的に書けます。 [Rust Playground](#) というサイトでは Rust プログラムを手軽に試すことができます。試しに下の内容を実行してみても構いません。サイトを開いたら最初から似たようなプログラムが入力されているかもしれません。

```
fn main() {  
    print!("hello world!")  
}
```

# コメント

- Rust のコメントには、一行コメント (`//`) と、ブロックコメント (`/*`) (`*/`) があります。 `/*` ブロックコメントは `/*` このようにネストして `*/` 書くことができます。 `*/`



# 式と文

- Rust は式ベースの言語です．ほとんどが**式**（**expression**）で表されます．ここで式は返り値を評価するものです．簡単に言うと，式は何かしらの値を返します．それに対して，**文**（**statement**）は処理を実行しますが値を返しません．

# ブロック

- 式の1つに**ブロック**があります。これは `{` と `}` で囲んだものです。例えば `{0}` というのは `0` を返す式です。ブロックが返す値は省略することができます。その場合は `()` を返します。この `()` を**ユニット**と言います。つまり、`{}` というのは `{()}` ということになります。
- ブロックには2つの機能があります。1つはスコープを作成します。もう1つは、文をいくつも記述できることです。

```
{ statement; statement; statement; ...; (expression) }
```

ここでセミコロン (`;`) は式を文に変化させるものです。

# オブジェクト

- 数値や関数や参照など、型の実体はすべて**オブジェクト**です。つまり、式が返す値もまたオブジェクトになります。例えば、`1` という値も数値オブジェクトであり、`1 == {1}` という関係にあります。

# 所有権

- オブジェクトには**所有権** (Ownership) が付いています。この所有権には2つの属性があります。

所有権		オブジェクト	
原本 / 仮	不変 / 可変		

# 束縛

- `let` 文を使うことでオブジェクトと変数を**束縛**します。変数はそのスコープから外れたときに束縛していた所有権を放棄します。また、最初に束縛したオブジェクトの所有権は基本的に**原本**となり、原本および仮の所有権がすべて放棄された時にオブジェクトは破棄されます。



# 参照

- Rust では所有権を使ってオブジェクトを受け渡します。通常は所有権を渡してしまうと束縛が解除されて、受け取った側がそれを束縛します。そこで、**仮**の所有権を作成して相手に渡すことで、渡す側は束縛を解除されず、仮の所有権を受け取った側はその所有権を使ってオブジェクトを操作することが出来ます。そして、受け取った側の変数がスコープを外れた時に束縛していた仮の所有権が破棄されます。この時、原本または他の仮の所有権があればオブジェクトは破棄されません。仮の所有権を作成する方法の1つが**参照** (**reference**) です。これは `&` 演算子を使います。

# 可変性

- Rust は標準でオブジェクトが**不変**（immutable）です。そこで、束縛時に `mut` キーワードをつけることで**可変**（mutable）にすることができます。

# 束縛・参照・可変のまとめ

- 今までのことをまとめると次のようになります。

`let a = object`

変数 a

原本 不変 所有権

オブジェクト

`let mut a = object`

変数 a

原本 可変 所有権

オブジェクト

`let a = &object`

変数 a

仮 不変 所有権

オブジェクト

`let a = &mut object`

変数 a

仮 可変 所有権

オブジェクト



# 変数から参照の作成

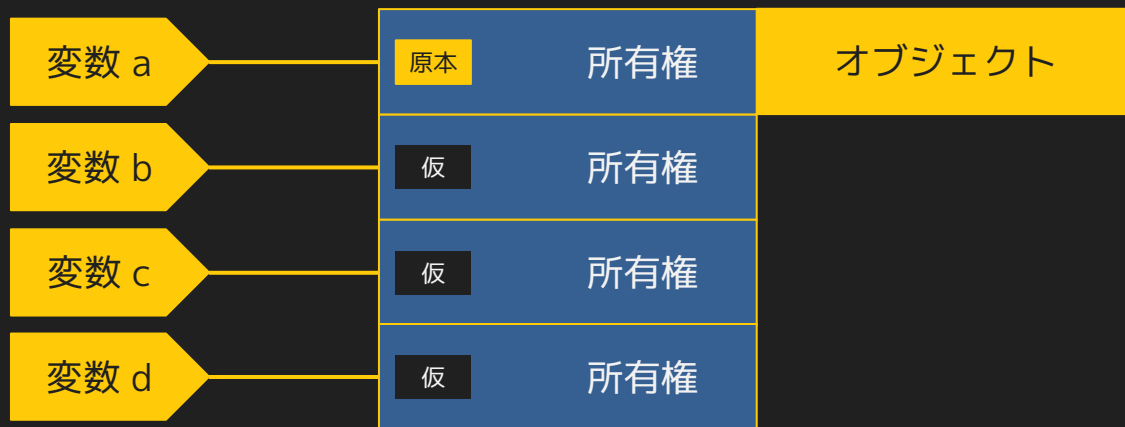
- 原本の所有権から仮の所有権を作成することが出来ます。また、仮の所有権を複製することも出来ますし、仮の所有権からさらに仮の所有権を作れます。ただし、仮の所有権から原本の所有権は作れません。

```
let a = object
```

```
let b = &a
```

```
let c = b
```

```
let d = &c
```



# 借用チェック

- 同一オブジェクトに対する参照と可変について、いくつか制限があります。
  - 不変参照 (`&`) は何個でも同時に存在することが出来る
  - 不変参照 (`&`) と可変参照 (`&mut`) は同時に存在することが出来ない
  - 可変参照 (`&mut`) は同時に1つしか存在することが出来ない
- ここで大事なことは、上記の制限は**関数呼び出し時**（かつコンパイル時）にチェックされるということです（これを**借用チェック**と呼びます）。このチェックが行われる直前の可変参照（必ず1つ）もしくは不変参照（複数可）がその時に存在していることになります。少なくとも可変参照を作成した時には、それまでの不変参照または可変参照がすべて無効となり、存在しないことになります。もちろん、あくまで同一オブジェクトに対する参照に対してです。

# 借用チェック

```
fn main() {  
    let a = 10;           // immutable object  
    let aref1 = &a;       // reference  
    let aref2 = &a;       // reference  
    println!("{}", a, aref1, aref2); // borrow check!! - OK  
}
```

# 借用チェック

```
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;  // mutable refernece  
    *a_mut_ref2 = 20;         // assign  
    println!("{}", a);        // borrow check!! - OK  
}
```

# 借用チェック

```
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  
    *a_mut_ref1 = 20;         // assign (error)  
    println!("{}", a);        // borrow check!! - Error!  
}
```

# 借用チェック

```
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  
    println!("{}", a_ref1);  // borrow check!! - Error!  
}
```

# 借用チェック

```
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  
    //println!("{}", a_mut_ref2);      // borrow check!! - Error!  
    //println!("{}", a_ref1, a_ref2);  // borrow check!! - Error!  
    println!("{}", a_ref2);          // borrow check!! - OK  
}
```

# 借用チェック

- 関数呼び出しによる借用チェックによって、スコープから抜けていない変数であっても、それが参照なら存在していないことになりうるということです（ここで存在していないと言っていますが、実際には存在できないようにコンパイル時にエラーが出るということ）。参照を束縛した変数になるべく作らないことが大切です。



# 参照外し

- 参照（仮の所有権）を使ってオブジェクトの操作をする場合は**参照外し**（**dereference**）が必要です。これは `*` 演算子を使います。

```
fn main() {  
    let mut a = 10;           // mutable object  
    let a_mut_ref = &mut a;   // mutable reference  
    *a_mut_ref = 20;          // dereference and assign  
    println!("{}", a_mut_ref); // auto dereference  
}
```

参照外しは関数に渡した時や、`.` 演算子によるフィールド操作・メソッド呼び出し時などにおいて自動で行われる場合があります。

# コピートレイト

- Rust には**トレイト** (**trait**) というデータ型を分類する概念があります。例えば、数値全般を表す `Num` というトレイトがあったとき、それを実装しているデータ型はすべて数値型として分類することができる、というものです。トレイトには特有のメソッドを実装することが出来ます。また、ジェネリクスにおいて、あるトレイトを実装した型であるという制約をかけることが出来ます。これを**トレイト境界** (**trait bound**) と呼びます。

# コピートレイト

- トレイトは標準でいくつか実装されているものがあり、その1つが**コピートレイト** (Copy Trait) です。束縛したオブジェクトがコピートレイトを実装したデータ型の変数から別の変数に束縛するときは、所有権は移動せず、値をコピーして新しいオブジェクト（そして所有権）を作成します。Rust のプリミティブ型はコピートレイトを実装しています。

```
fn main() {  
    let a = 10;           // immutable object  
    let b = a;            // copy  
    print!("{}", a, b); // borrow check!! - OK  
}
```

# コピートレイト

- 不変参照 (&) もコピートレイトを実装しています.

```
fn main() {  
    let a = 10;                // immutable object  
    let a_ref = &a;            // reference  
    let a_ref_copy = a_ref;    // copy reference  
    print!("{}", a, a_ref, a_ref_copy); // borrow check!! - OK  
}
```

# コピートレイト

- 注意なのが、可変参照 (`&mut`) はコピートレイトを実装していません。なぜなら、可変参照は1つしか存在してはいけないからです。

```
fn main() {  
    let mut a = 10;           // mutable object  
    let a_mut_ref = &mut a;   // mutable reference  
    let a_mut_ref_move = a_mut_ref; // move mutable reference  
    print!("{}", a_mut_ref);   // borrow check!! - Error!  
}
```

```
fn main() {  
    let mut a = 10;           // mutable object  
    let a_mut_ref = &mut a;   // mutable reference  
    let a_mut_ref_move = a_mut_ref; // move mutable reference  
    print!("{}", a_mut_ref_move); // borrow check!! - OK  
}
```

# コピートレイト

- データ型がコピートレイトを実装しているかどうかはドキュメントに記載されています。また、下記に示す関数 `copy_trait_check` では、トレイト境界を使って引数の型がコピートレイトを実装していることを強制します。実装されていなかったらコンパイルエラーになります。エラーから分かるように、`String` 型はコピートレイトを実装していません。

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

fn main() {
    let s = String::from("hello");    // String
    copy_trait_check(s);
    // ^ the trait `Copy` is not implemented for `String`
    // error[E0277]: the trait bound `String: Copy` is not satisfied
    let a = 10;                       // i32
    copy_trait_check(a); // OK
}
```

# 不変束縛から可変束縛

- 不変束縛の変数から可変束縛の変数に変えることができます。

```
let a = object
```



```
let mut b = a
```



変数 **a** から変数 **b** に所有権が移動し，可変に変わります．そして，変数 **a** の束縛は解除されます．もし，オブジェクトがコピートレイトを実装していたらコピーが作成され，変数 **a** はオブジェクトを束縛したままで，変数 **b** には新しい可変のオブジェクトが束縛されます．

# データ型

- Rust の標準にある基本的なデータ型は次のとおりです：

- スカラー型

- 整数型：`i8`, `u8`, `i16`, `u16`, `i32`, `i64`, `u64`, `isize`, `usize`
- 浮動小数型：`f32`, `f64`
- ブーリアン型：`bool`
- 文字型：`char`

- 複合・配列型

- タプル型：`(500, 6.4, true)`. `()` はユニット.
- 配列型：`[1, 2, 3, 4, 5]`, `[3; 5]` = `[3, 3, 3, 3, 3]`

- 数値型のリテラルには次のものが使えます：

`98_222`(10進数), `0xff`(16進数), `0o77`(8進数), `0b1111_0000`(2進数), `b'A'`(バイト), `0.`(浮動小数)



# データ型

- 基本的なデータ型はコピートレイトを実装しています。また、複合・配列型については、含まれている要素がすべてコピートレイトを実装していれば、全体もコピートレイトを実装したことになります。
- 参照も型の1つで、不変参照はコピートレイトを実装していますし、可変参照は実装していません。また、参照のまた参照ということも可能です。

```
fn main() {  
    let a = 42;  
    let ref_ref_ref_a = &&&a;  
    let ref_a = **ref_ref_ref_a;  
    let b = *ref_a;  
    print!("{}", a, b);  
}
```

# データ型

- 比較するときは基本的に同じ型でなくてはならないので、参照もまた型であるということが以下でわかります。

```
fn main() {  
    let a          = 10;      // immutable object  
    let a_ref      = &a;      // reference  
    let a_ref_ref  = &a_ref;  // reference to reference  
    println!("{}", a == a_ref);  
    // error[E0277]: can't compare `{integer}` with `&{integer}`  
    println!("{}", a_ref_ref == a_ref);  
    // error[E0277]: can't compare `&{integer}` with `{integer}`  
}
```

# データ型を指定した束縛


- Rust は強い型推論を持っていますが、意図的にデータ型を指定したい場合があります。その場合は変数名の後ろに `:` を付けてデータ型を指定します。

```
fn main() {  
    let a: i32 = 10;  
    let b: u32 = 20;  
    let c: f32 = 0.;  
    let d: &i32 = &50;  
    print!("{}", a, b, c, d);  
}
```

# 要素を分解して束縛

- 変数は**パターン**を使って、要素を分解して束縛することができます.

```
fn main() {  
    let (x,y,z) = (1,2,3);  
    let [a,b,c] = [4,5,6];  
    let (i,_,_) = (7,8,9);  
    println!("xyz= {} {} {}", x, y, z);  
    println!("abc= {} {} {}", a, b, c);  
    println!("  i= {}", i);  
}
```

 は**ワイルドカード**と呼ばれるもので、オブジェクトを無視するときに使います.

# 同じ変数名の束縛

- Rust では同じスコープ内で、変数名を使い回すことができます。

```
fn main() {  
    let str_len = String::from("hello world!");  
    let str_len = str_len.len();  
    println!("{}", str_len);  
}
```

# シャドーイング

- あるスコープのさらにローカルなスコープにおいても外側にある変数名と同じ名前で新しく束縛できます。このとき、外側の変数はローカルから隠れます。これを**シャドーイング**と言います。

```
fn main() {  
    let a = 10;  
  
    { // local scope  
        let mut a = 20;  
        a += 30;  
        println!("{}", a); // 50  
    }  
  
    println!("{}", a); // 10  
}
```

# 型変換

- 明示的に数値オブジェクトを型変換して使いたい場合があります。その場合は `as` を使います。

```
fn main() {  
    let a = 13u8;  
    let b = 7u32;  
    let c = a as u32 + b;  
    println!("{}", c);  
  
    let t = true;  
    println!("{}", t as u8);  
}
```

# 関数

- 関数を定義するには `fn` を使い、本体は `{}` で囲みます。引数の型は必ず明記しなければなりません。 `fn` は文で、 `{}` は式です。式は返り値を持つものでしたよね。返り値の型は `->` で指定します。  
また、 `return` で処理を中断して値を返すことができます。

```
fn add(a: i32, b: i32) -> i32 {  
    a+b  
}  
  
fn main() {  
    print!("{}", add(10,20));  
}
```



# 関数の引数で分解束縛

- 関数の引数に対してパターンによる分解束縛をすることができます.

```
fn print_coordinates(&(x, y): &(i32, i32)) {  
    println!("location: ({}{})", x, y);  
}  
  
fn main() {  
    let point = (3, 5);  
    print_coordinates(&point);  
}
```

# if 式

- `if` は条件によって処理を分岐するものです. `if`, `else`, `else if` が使えます. それぞれに続くものは式 `{}` です.

```
fn main() {  
    let number = 6;  
    if number % 4 == 0 {  
        println!("number is divisible by 4");  
    } else if number % 2 == 0 {  
        println!("number is divisible by 2");  
    } else {  
        println!("number is not divisible by 4 or 2");  
    }  
}
```

# if 式

- `if` は式なので、値を返せます。ただし、その場合は返す値が同じ型でなければなりません。

```
fn main() {  
    let condition = true;  
    let number = if condition { 5 } else { 6 };  
    // let number = if condition { 5 } else { "six" }; Error!  
    println!("The value of number is: {}", number);  
}
```

# loop 式

- 無限ループするには `loop` を使います. ループから抜ける場合は `break` を使います. `loop` もまた式なので, `break` に返り値を指定することができます.

```
fn main() {  
    let mut counter = 0;  
    let result = loop {  
        counter += 1;  
        if counter == 10 {  
            break counter * 2;  
        }  
    };  
  
    println!("The result is {}", result);  
}
```

# while 式

- 条件を満たしている間だけループさせる場合は `while` 式を使います。  
`while` 式は常に `()` を返します。 `break` は使えますが、値を返すことは出来ません。

```
fn main() {  
    let mut number = 3;  
    while number != 0 {  
        println!("{}", number);  
        number -= 1;  
    };  
  
    println!("LIFTOFF!!!");  
}
```

# for 式

- イテレータを使って、各要素に対して処理を行いたい場合は `for` を使います。 `for` 式は常に `()` を返します。（イテレータとは、連続する一連のデータへのアクセスを提供するオブジェクトのことです）

```
fn main() {  
    let a = [10, 20, 30, 40, 50];  
  
    for element in a.iter() {  
        println!("the value is: {}", element);  
    }  
}
```

# スライス

- **スライス**は参照の1つで、別のオブジェクト内の連続した要素を指し示すものです。スライスを取得するには、オブジェクトに対して**数列指定** (`[m..n]`) します。参照なので、スライスの型は例えば配列だと `&[T]` となります。`T` は任意の型です。

```
fn main() {  
    let a = [1,2,3,4,5];  
    let a_slice = &a[1..3];  
    dbg!(a_slice); // [2,3]  
}
```

# 数列指定

- Rust はインデックスが `0` から始まります． 数列指定では開始インデックスと終了インデックスを `..` を使って指定します． 例えば `m..n` なら `[m, m+1, m+2, ..., n-1]` となります． `m..=n` の場合は `[m, m+1, m+2, ..., n]` となります． 開始インデックスと終了インデックスは省略することが出来ます．

```
let s = String::from("hello");
let slice = &s[0..2];
let slice = &s[0..=2];
let slice = &s[..2];
let slice = &s[3..s.len()];
let slice = &s[3..];
let slice = &s[..];
```



# 文字列リテラル

- 文字列のスライスの型は `&str` です。そして、文字列リテラルは不変の文字列スライス (`&str`) です。

```
let s = "Hello, world!";
```

# 構造体

- **構造体**はデータ型の要素を集めたものです。1つ1つの要素を**フィールド**と呼びます。構造体の定義は `struct` を使い、フィールドは名前と型を指定します。

```
struct User {  
    username: String,  
    email: String,  
    sign_in_count: u64,  
    active: bool,  
}
```

# 構造体

- 構造体のオブジェクトを作成する場合は、各フィールドを `key: value` という形で束縛します.

```
let user1 = User {  
    email: String::from("someone@example.com"),  
    username: String::from("someusername123"),  
    active: true,  
    sign_in_count: 1,  
};
```

# 構造体

- ここでは「**オブジェクト**」と「**インスタンス**」について説明します。一般的に構造体や列挙型など（オブジェクト指向でのクラス）の実体は「**インスタンス**」と呼ばれています。しかし、本書ではそれらを「**オブジェクト**」に統一します。そして、「**インスタンス**」は、関数型プログラミング言語 Haskell に従って、**データ型**を表します。例えば、コピートレイトを実装した構造体 `Hoge` があるとします。このとき、`Hoge` はコピートレイトの**インスタンス**（**データ型**）です。そして、トレイト境界でコピートレイトを指定した場合、コピートレイトのインスタンスである `Hoge` は制約を満たしていることになります。

# 構造体

- 可変のオブジェクトを作成するとすべてのフィールドが可変になります。オブジェクトのフィールドは `.` 演算子を使って指定します。

```
let mut user1 = User {  
    email: String::from("someone@example.com"),  
    username: String::from("someusername123"),  
    active: true,  
    sign_in_count: 1,  
};  
  
user1.email = String::from("anotheremail@example.com");
```

# 構造体

- オブジェクト作成時に指定する変数名と構造体のフィールド名が一致している場合、フィールド名を省略することができます。

```
fn build_user(email: String, username: String) -> User {  
    User {  
        email,  
        username,  
        active: true,  
        sign_in_count: 1,  
    }  
}
```

# 構造体

- あるオブジェクトのフィールドに束縛したものを使って、新しいオブジェクトを作成するときに便利な構文があります。オブジェクト作成時に、明示的にフィールドを指定しなかったものは `..` の後に渡したオブジェクトのフィールドを束縛します。ただし、コピートレイトを実装している型なら複製され、そうでないなら所有権が移動することに注意が必要です。

```
let user2 = User {  
    email: String::from("another@example.com"),  
    username: String::from("anotherusername567"),  
    ..user1  
};
```

# タプル構造体

- 指定した要素で構成されたタプルに名前をつけることができます。このようなタプルを**タプル構造体**といいます。この場合、フィールド名はありません。タプル構造体は同じ構成をしていても別の型として区別されます。タプルは `.0` というように要素のインデックスを指定するか、要素の分解を使います。

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let Point (x,y,z) = Point(0, 0, 0);

    println!("{}", black.0, black.1, black.2);
    println!("{}", x, y, z);
}
```



# Newtype

- Rust は静的型付け言語です。この特性を利用して既存の型から新しい型を作成することで意図的な意味を付加させて、制約することができます。また、既存の型を利用するので薄いラッパー型と考えることも出来ます。これは Newtype パターンと呼ばれるもので、タプル構造体を利用する例の1つです。

# Newtype

- 例えば, `String` 型から `Password` 型を作成し, `{}` で出力したときに伏せ字にする場合は次のようになります.

```
use std::fmt;

struct Password(String);

impl fmt::Display for Password {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.0.chars().map(|_| '*').collect::<String>())
    }
}

fn main() {
    let a = String::from("123456789");
    println!("{}", a); // 123456789

    let a = Password(String::from("123456789"));
    println!("{}", a); // *****
}
```

# ユニット構造体

- `()` のことをユニットと言いました．このようなフィールドを何も持たない構造体のことを**ユニット構造体** (Unit-like Structs) と言います．（日本語ドキュメントだとユニット様構造体と翻訳されていますが，ユニット構造体の方が言いやすいし意味も伝わるでしょう）．ユニット構造体はフィールドを持たず，トレイトだけ実装するといった時に使われるようです．

# メソッド

- **メソッド**は関数に似ていますが、構造体と関連していて、`self` を使うことで、そのメソッドを呼び出したオブジェクトを操作することが出来ます。メソッドの第一引数は必ず `self` になります。また、基本的に不変参照 (`&self`) か可変参照 (`&mut self`) になります。もちろん、可変参照のメソッドは、呼び出し元が可変の所有権を使って呼び出さなければなりません。メソッドは `impl` ブロックの中で、関数と同じく `fn` を使って定義します。

# メソッド

```
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
impl Rectangle {  
    fn area(&self) -> u32 {  
        self.width * self.height  
    }  
}  
  
fn main() {  
    let rect1 = Rectangle { width: 30, height: 50, };  
  
    println!("The area of the rectangle is {} square pixels.", rect1.area());  
}
```

# メソッド

- **[重要]** メソッドの第一引数が参照ではなく `self` の場合があります。これは呼び出し元のオブジェクトの所有権をメソッドが受け取ります。つまり、このメソッドを呼び出したとき、呼び出し元のオブジェクトが束縛されていたら、それは解除され使用できなくなるということです。これはメソッド呼び出しによってオブジェクトが別のものに変換するといったときに使われるようです。例えば、後に出てくる `Option` や `Result` の `unwrap` というメソッドは `unwrap(self)` です。

# 関連関数

- `impl` ブロックの中で関数を定義することができます。それは `self` を引数に取りません。このような関数を**関連関数**と呼びます。これは構造体に関連しているにも関わらず、そのオブジェクトが無くても呼び出すことができます。関連関数は主にその構造体のオブジェクトを生成する関数の定義に使い、そのような関連関数には `new` という名前が使われます。関連関数は構造体の名前に `::` を使って呼び出します。

# 関連関数

```
struct Point {  
    x: f64,  
    y: f64,  
}  
  
impl Point {  
    fn new(x: f64, y: f64) -> Self { // Self は実装している型の型エイリアス  
        Self { x, y }  
    }  
}  
  
fn main() {  
    let a = Point::new(3., 5.);  
  
    print!("x={}, y={}", a.x, a.y);  
}
```



# impl ブロック

- `impl` ブロックは複数定義することができます。トレイトごとに実装を分けたりすることができます。

```
impl Rectangle {  
    fn area(&self) -> u32 {  
        self.width * self.height  
    }  
}  
  
impl Rectangle {  
    fn can_hold(&self, other: &Rectangle) -> bool {  
        self.width > other.width && self.height > other.height  
    }  
}
```

# 列挙型

- **列挙型**は取りうる様々な値を列挙しておき、そのうちのどれか1つだけ値を取るデータ型です。列挙した値のことを**列挙子** (**variant**) と呼びます。構造体がフィールドの集合に対して **AND** の関係であると考えれば、列挙型は **OR** の関係にあると言えます。列挙型は **enum** を使って定義し、列挙子は **::** で指定します。

```
enum IpAddrKind {  
    V4,  
    V6,  
}  
  
let four = IpAddrKind::V4;  
let six = IpAddrKind::V6;
```

# 列挙型

- 列挙子はそれぞれ別々の型にすることが出来ます.

```
enum IpAddr {  
    V4(u8, u8, u8, u8),  
    V6(String),  
}  
  
let home = IpAddr::V4(127, 0, 0, 1);  
let loopback = IpAddr::V6(String::from("::1"));
```

- 列挙型は、構造体のようにメソッドを定義することも出来ますし、トレイトのインスタンスにもなることが出来ます.

# ジェネリクス

- Rust では、例えば数値演算をするときに、左値と右値の型が同じである必要があります。ここで、加算を行う `add` という関数を考えてみます。数値型には整数型や浮動小数点型などあります。型の数だけ `add` 関数を定義してしまうと同じコードが大量に出来てしまいます。そこで、引数の型が変わっても関数本体のコードが変わらない場合は、任意の型を受け取れる関数を定義することでコードの重複を避けることが出来ます。このような仕組みを**ジェネリクス**と言い、任意の型のことを**ジェネリック型**と言います。

# ジェネリクス

- 関数，構造体，列挙型でジェネリック型を使うには，それぞれの名前の後ろに `<>` でジェネリック型の名前を指定します．名前には慣例的に `T` をよく使います．また，複数であれば，`,` で列挙します．

```
fn add<T>(a: T, b: T) -> T {  
    a+b  
}  
  
struct Point<T> { x: T, y: T }  
  
enum Result<T,E> {  
    Ok(T),  
    Err(E),  
}
```

# ジェネリクス

- メソッドの定義では次のように記述します.

```
struct Point<T> { x: T, y: T }  
  
impl<T> Point<T> {  
    fn xy(self) -> (T, T) {  
        (self.x, self.y)  
    }  
}
```

`impl` の後ろに `<T>` を宣言しています. こうすることで `Point<T>` の `T` がジェネリック型であることを明示しています. もし, `impl<T>` でなければ, `Point<T>` の `T` はジェネリック型ではなく `T` という名前の型を指定することになってしまいます.

# ジェネリクス

- 1 つ前で, `impl<T>` を定義することでジェネリック型の `T` を使うことを明示していることがわかりました. これとは逆に, ジェネリック型に明示的な型を指定するやり方があります.

```
impl Point<f64> {  
    fn distance(&self) -> f64 {  
        (self.x.powi(2) + self.y.powi(2)).sqrt()  
    }  
}
```

ジェネリック型は任意の型を受け取りますが, 静的型付け言語では, コンパイル時に型がわかるので, 型に特化したコードが生成されます. このような仕組みを**単相化** (monomorphization) と言います.

# ジェネリクス

- Rust は強い型推論があるので、ジェネリック型に対して適切な型を自動で推論してくれます。しかし、明示的に指定したい場合もあります。この場合は `::<...>` 演算子を使います。この演算子は魚が速く泳いでいるように見えることから **turbofish** と呼ばれています。

```
let point = Point::<f64>{x: 3., y: 5.};
```

::<>

::<>

::<>

::<>

::<>

::<>

::<>

::<>

::<>

::<>

::<>

::<>



# Option

- Rust には無効な値を取ることができる便利な列挙型として `Option` があります. `T` はジェネリック型です.

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

# Option

- `Option` 型に有効な値を束縛するときは `Some` を使います. また, 無効な値を束縛するときは `None` を使います.

```
let some_number = Some(5);  
let some_string = Some("a string");  
let absent_number: Option<i32> = None;
```

`Option` 型のオブジェクトから値を取り出すには, この後に説明するパターンマッチングか, `unwrap` などのメソッドを使います.

# match 式

- **パターンマッチング**は、式の値がパターンに一致するかしないかを判定する仕組みです。 `match` 式は、パターンマッチングによって評価する式を変えるときに使います。

```
enum Coin {  
    Penny,  
    Nickel,  
    Dime,  
    Quarter,  
}  
  
fn value_in_cents(coin: Coin) -> u8 {  
    match coin {  
        Coin::Penny => 1,  
        Coin::Nickel => 5,  
        Coin::Dime => 10,  
        Coin::Quarter => 25,  
    }  
}
```

# match 式

- `match` 式はパターンと式を `=>` で結合したものを並べたものです. この `パターン => 式` のことを**アーム** (arm) と言います. `match` 式はアームのパターンを順番に処理していき, 最初にパターンに一致した式を評価してその結果を返します. そして, パターンに一致したアーム以降は処理されません. このような仕組みを**短絡評価**または**ショートサーキット**と呼びます

```
fn plus_one(x: Option<i32>) -> Option<i32> {  
    match x {  
        None => None,  
        Some(i) => Some(i + 1),  
    }  
}  
  
let five = Some(5);  
let six = plus_one(five);  
let none = plus_one(None);
```

# match 式

- `match` 式はマッチング対象のオブジェクトが取りうる値をすべて網羅しなければなりません。そのため、記述したアーム以外に一致する**ワイルドカード** (`_`) を使うことができます。

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

# match 式

- `Option` 型が束縛しているオブジェクトを `match` 式で取り出すことができます。ここで注意なのが、`match` 式でパターンが一致したときに、束縛しているオブジェクトを受け取りますが、そのとき所有権も移動しているということです。

```
fn main() {  
    let a: Option<String> = Some(String::from("hello"));  
    match a {  
        Some(x) => println!("{}", x), // move ownership  
        None => ()  
    }  
    println!("{}", a);  
    //           ^ value borrowed here after partial move  
    // error[E0382]: borrow of partially moved value: `a`  
}
```

# match 式

- これは何が起きているかと言うと、変数 `a` が束縛している `Option` 型のオブジェクトが、内部で束縛しているオブジェクトの所有権をアームのパターンによって取り出され所有権が渡されています。これにより、変数 `a` は束縛したままですが、その内部では何も束縛していないこととなります。なので、**部分移動**（**partial move**）が発生していることになりエラーとなります。この部分移動に対応する方法として次の2つがあります。

# match 式

- 1 つはアームのパターンでオブジェクトを参照で受け取る方法です。注意なのが、パターンで参照を取得するときは `ref` を使います。可変参照なら `ref mut` です。

```
fn main() {  
    let a: Option<String> = Some(String::from("hello"));  
    match a {  
        Some(ref x) => println!("{}", x), // reference  
        None => ()  
    }  
    println!("{:?}", a); // borrow check!! - OK  
}
```



# match 式

- もう1つは, 返り値としてオブジェクトを返すことです.

```
fn main() {  
    let a: Option<String> = Some(String::from("hello"));  
    let a = match a {  
        Some(x) => { println!("{}", x); Some(x) },  
        None => None,  
    };  
    println!("{:?}", a); // borrow check!! - OK  
}
```

# 再び分解束縛

- 変数に束縛するときにパターンを使って分解出来ることを覚えていますか？  
この分解束縛のパターンでも参照を使うことが出来ます。

```
struct Account { name: String, pass: String }

fn main() {
    let a = Account { name: String::from("name"), pass: String::from("pass") };
    let Account { name, pass } = a;    // move ownership
    println!("{}", name, pass);        // borrow check!! - OK
    println!("{}", a.name, a.pass);    // borrow check!! - Error
}
```

```
let Account { ref name, ref pass } = a; // reference
println!("{}", name, pass);             // borrow check!! - OK
println!("{}", a.name, a.pass);         // borrow check!! - OK
```

# if let 式

- `match` 式のアーム（ワイルドカード以外）が1つのときは `if let` 式を使うと短く記述することが出来ます。

```
let some_u8_value = Some(0u8);  
match some_u8_value {  
    Some(3) => println!("three"),  
    _ => (),  
}
```



```
if let Some(3) = some_u8_value {  
    println!("three");  
}
```

`if let` と同様に `while let` も使うことが出来ます。

# マッチガード

- **マッチガード**は `match` 式のアームのパターンに、さらに `if` 条件を加えることができます。これにより、より複雑なパターンを扱えます：

```
let num = Some(4);

match num {
  Some(x) if x < 5 => println!("less than five: {}", x),
  Some(x) => println!("{}", x),
  None => (),
}
```

# エラー処理

- 基本的に復帰不能なエラーが発生したら、どうしようも出来ません。メッセージを表示してプログラムを終了する手っ取り早い方法が `panic!` です

```
fn main() {  
    panic!("crash");  
}
```

# Result

- どこかでエラーが発生したとしても、すぐにプログラムを終了させるわけにはなかなかいきません。ある関数の内部でエラーが発生したら、それを呼び出し元に知らせる必要があります。そこで `Result` 型が使われます。

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

# Result

- ここではファイル処理を考えてみます。既存のファイルを開いて処理をしたいとします。もし、ファイルが無ければ作成します。その場合、最初にファイルを開こうとしたときにエラーが発生し、そのエラーがファイルが無かったことを表していればファイルを新規に作成するようにします。ファイルを開く処理 `File::open` は `std::io::Result` 型を返します。これは `Result<T, Error>` 型の別名です。

# Result

```
use std::{fs::File, io::ErrorKind};

fn main() {
    let f = File::open("hello.txt");
    let f = match f {
        Ok(file) => file,
        Err(ref error) if error.kind() == ErrorKind::NotFound => {
            match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Tried to create file but there was a problem: {:?}", e),
            }
        },
        Err(error) => {
            panic!("There was a problem opening the file: {:?}", error)
        },
    };
}
```



# unwrap, expect

- `Option` 型, `Result` 型ともに, 値を取り出す `unwrap` という関数があります. これは, もし値が `None`, `Err` のときに, `panic!` を呼び出します.

```
pub fn unwrap(self) -> T
```

- `unwrap` が `panic!` を呼び出すと, 標準のエラーメッセージが表示されますが, `unwrap` の代わりに `expect` を呼ぶと, エラーメッセージに情報を追加することが出来ます.

```
pub fn expect(self, msg: &str) -> T
```

# unwrap

- `Option` 型と `Result` 型の `unwrap` は以下のように `match` 式を短くしたものです.

```
option.unwrap()
```



```
match option {  
    Some(v) => v,  
    None => panic!(...),  
}
```

```
result.unwrap()
```



```
match result {  
    Ok(v) => v,  
    Err(e) => panic!(...),  
}
```

# エラー伝搬

- `Option` 型, `Result` 型を返す関数の中で, 値が `None` または `Err` のときに, 処理を中断して呼び出し元に値を返す仕組みが用意されています. それは `?` 演算子を使います.

```
fn hoge() -> Option<i32> {  
    let a = Some(10);  
    let b = a?;  
    Some(b)  
}
```

```
fn hoge() -> Option<i32> {  
    let a = None;  
    let b = a?; // return Option<i32>::None  
    Some(b)  
}
```

# コンビネータ

- `Option` 型, `Result` 型も**コンビネータ**です. このコンビネータがエラー処理のコードを大幅に削減してくれます. 手続き型であれば, 1つ1つの関数呼出しの結果がエラーか無効な値かを確認していきます. これだと, 確認コードが大量に出来てしまいます. そこで, まずはエラー伝搬です. 関数が `Option` か `Result` を返せば, `?` 演算子を使ってチェーン方式で処理を記述することが出来ます. 途中でエラーが発生すれば, 処理を打ち切ってエラー伝搬されます.

```
let ret = open()?.read()?.replace()?.write()?.close()?;
```

# コンビネータ

- コンビネータとは簡単に言うと、高階関数のことで、**高階関数**とは関数を引数に取る関数のことです。例えば、関数  $f(x)$  と  $g(x)$  , これらの合成関数が  $(f \circ g)(x) = f(g(x))$  とします。この場合、この  $\circ$  がコンビネータで、2つの関数を取っています。先程の `open.read.write.close` で考えてみると `close(write(read(open())))` の関係に見えないでしょうか。ここで `.` 演算子がコンビネータであり、その役を担っているのが `Option` 型と `Result` 型と考えることができます。

# コンビネータ

- `Option` 型, `Result` 型にはコンビネータとしての便利なメソッドが多く用意されています. 基本的なものとして, `map` は値に関数を適用して, その結果をコンビネータに変換します.

```
pub fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Option<U>      // Option
pub fn map<U, F: FnOnce(T) -> U>(self, op: F) -> Result<U, E> // Result
```

`and_then` は関数を適用して, その結果をそのまま返します. つまり, `and_then` に渡す関数はコンビネータを返します.

```
pub fn and_then<U, F: FnOnce(T) -> Option<U>>(self, f: F) -> Option<U>      // Option
pub fn and_then<U, F: FnOnce(T) -> Result<U, E>>(self, op: F) -> Result<U, E> // Result
```

# コンビネータ

- コンビネータは型を合わせる必要があります。 `Option` のメソッドに渡す関数は、単純に `T` 型を返す関数や `Option` を返す関数ならよいのですが、`Result` を返す場合にはそのままでは利用できません。  
そこで、`Option` と `Result` には相互に変換するメソッドがいくつかあります。例えば、`ok_or` は `Option` から `Result` に、`ok` は `Result` から `Option` に変換します。これにより `Option` や `Result` を返す関数を1つのメソッドチェーン内に利用することが出来ます。

# コンビネータ

- コンビネータは `?` 演算子を使っていなければ、途中の処理で `None` になったり、`Err` になった場合、チェーンの最後の型で返ってきます。これによりエラー処理を書く場所が少なくなります。
- メソッドのところでも少し触れましたが、コンビネータのメソッドの引数は `self` が多いです。これは、メソッド呼び出しで、`Option<u32>` が `Option<f32>` になったり、`Option<T>` が `Result<T,E>` になったり型の変換を行っているからです。



# トレイト

- すでにトレイト, コピートレイト, トレイト境界について触れていますが, ここでさらに詳しく解説します. まずは, おさらいです. **トレイト (trait)** はデータ型を分類する仕組みのことです. また, ジェネリック型に**トレイト境界**を指定することで, その型が特定のトレイトの**インスタンス**であることを強制します. さらに, トレイトには特有のメソッドを実装することが出来, 型に対して共通の振る舞いを定義することが出来ます. つまり, あるトレイトのメソッドは, そのインスタンスであれば呼び出せることになります. また, インスタンスによってその振る舞いの実装を変えることも出来ます.

# トレイト

- トレイトを定義するには `trait` を使います。トレイト名を指定して、ブロック内に共通のメソッドを定義します。このメソッドはインスタンス側で実装しなければなりませんが、トレイト側で実装することも出来ます。この場合は、インスタンス側はトレイト側の実装をそのまま使うことも出来ますし、その振る舞いを上書き（**オーバーライド**）することも出来ます。

```
pub trait Geometry {  
    fn area(&self) -> f64;  
    fn name(&self) -> &str { return "Geometry" }  
}
```

# トレイト

- トレイトの実装は `impl A for B` で指定します. ここで `A` にはトレイト名を, `B` には実装する型を指定します.

```
impl Geometry for Rectangle {  
    fn area(&self) -> f64 {  
        self.width as f64 * self.height as f64  
    }  
    fn name(&self) -> &str { return "Rectangle" }  
}
```

# トレイト

```
pub trait Geometry {  
    fn area(&self) -> f64;  
    fn name(&self) -> &str { return "Geometry" }  
}  
  
struct Rectangle { width: u32, height: u32 }  
  
impl Geometry for Rectangle {  
    fn area(&self) -> f64 {  
        self.width as f64 * self.height as f64  
    }  
    fn name(&self) -> &str { return "Rectangle" }  
}  
  
struct Triangle { bottom: u32, height: u32 }  
  
impl Geometry for Triangle {  
    fn area(&self) -> f64 {  
        self.bottom as f64 * self.height as f64 * 0.5  
    }  
    fn name(&self) -> &str { return "Triangle" }  
}  
  
fn main() {  
    let a = Rectangle { width: 10, height: 20 };  
    let b = Triangle { bottom: 20, height: 5 };  
    println!("{}", a.name(), a.area());  
    println!("{}", b.name(), b.area());  
}
```

# トレイトの継承

- トレイトは別のトレイトのインスタンスになることができます。これを**継承**と呼ぶこともあります。 `trait 継承先 : 継承元` という形で指定します。継承したインスタンスは継承元のトレイトも実装する必要があります。

```
pub trait Geometry {  
    ...  
}  
  
pub trait Drawable: Geometry {  
    ...  
}  
  
impl Geometry for Rectangle {  
    ...  
}  
  
impl Drawable for Rectangle {  
    ...  
}
```

# トレイト境界

- トレイトのインスタンス型を表すには `impl A` のようにします. `A` はトレイト名です. 次の関数の引数 `geometry` は `Geometry` のインスタンスでなければなりません. これがトレイト境界です.

```
fn draw(geometry: impl Geometry) {  
    ...  
}
```

# トレイト境界

- トレイト境界の指定はより便利な方法があります：

```
fn draw<T: Geometry>(geom1: &T, geom2: &T) {  
    ...  
}
```

また、トレイトは `+` を使って複数指定することが出来ます：

```
fn draw(geometry: &(impl Geometry + Display))  
fn draw<T: Geometry + Display>(geometry: &T)
```

他にも `where` を使って次のように書くことも出来ます：

```
fn draw<T>(geometry: &T)  
    where T: Summary + Display
```

# トレイト境界

- ジェネリック型にもトレイト境界を指定することが出来ます。次のコードでは、`Display` と `PartialOrd` のインスタンスの場合なら、`cmd_display` メソッドが実装されます。

```
use std::fmt::Display;

struct Pair<T> { x: T, y: T }

impl<T> Pair<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}

impl<T: Display + PartialOrd> Pair<T> {
    fn cmp_display(&self) {
        if self.x >= self.y {
            println!("The largest member is x = {}", self.x);
        } else {
            println!("The largest member is y = {}", self.y);
        }
    }
}
```



# RAII

- **RAII (Resource Acquisition Is Initialization)** とはリソースの確保をオブジェクトの初期化時に行い, リソースの開放をオブジェクトの破棄と同時に行う手法です. これから解説する内容は公式だとスマートポインタと呼ばれていますが, 安易にポインタという用語を使うべきでないと思っているし, RAII はポインタに限った話でもないので本書では使いません.

# Deref トレイト

- 実は, Rust の参照は RAII の最も代表となる 1 つです. 参照は仮の所有権を保持し, スcopeから外れると仮の所有権を破棄します. 参照の機能は大きく 2 つあります. 参照先のオブジェクトを操作できること, 参照先のオブジェクトの仮の所有権を破棄することです. このうち, 参照先のオブジェクトを操作するには参照外しが必要です. これを実現しているのが `Deref` トレイトです. 参照外しは参照に対して `*` 演算子を使います. このように, RAII におけるリソースに対して操作をするには `Deref` トレイトを実装し `*` 演算子を使うことです. また, 可変参照に対しては `DerefMut` トレイトを実装します.

# Drop トレイト

- 参照のもう1つの機能が仮の所有権の破棄です。これは `Drop` トレイトで実装します。 `Drop` トレイトのインスタンスのオブジェクトは、それが破棄されるときに `drop` メソッドが呼ばれます。このメソッドでリソースの開放処理を行います。 `drop` メソッドは可変参照を引数に取ります。

```
impl Drop for Resource {  
    fn drop(&mut self) {  
        ...  
    }  
}
```

# Drop トレイト

- `drop` メソッドは基本的にオブジェクトが破棄されるときに自動で呼び出されますが、明示的に呼びたい場合があるかもしれません。その場合は、`std::mem::drop` 関数で強制的に呼び出してオブジェクトを破棄することが出来ます。ただし、あまり使うべきではありません。

# メモリ

- Rust では次の3つのメモリ領域があります。1つ目は**データメモリ**で、静的データが格納されています。静的データはプログラム実行中に存在するデータのことです。2つ目は**スタックメモリ**です。これは変数や関数呼び出し時の引数など一時的な格納場所で、コンパイラが最適化しやすく高速にデータ操作が出来ます。3つ目は**ヒープメモリ**です。ここにはプログラム実行中に利用できるメモリで、スタックメモリよりも大きなサイズを利用することが出来ます。ただし、利用するにはオーバーヘッドがかかります。また、スタックメモリのサイズはヒープメモリに比べてかなり限られているので、ヒープメモリを積極的に利用することになります。

# Box

- これまで、メモリを意識してきませんでした。基本的に作成したオブジェクトはスタックメモリに置かれます。また、`static` に指定したオブジェクトや文字列リテラルなどはデータメモリに置かれます。では、ヒープメモリに置くにはどうすればよいでしょうか。それが `Box<T>` です。使い方は簡単で、`Box::new` または `Box::<T>::new` にオブジェクトを渡すだけです

```
let a = Box::new(10);           // type inference
let a = Box::<i32>::new(20);    // explicit type
let a = 30;                     // immutable object
let b = Box::new(a);            // move object from stack memory to heap memory
let c = *b;                     // dereference
```

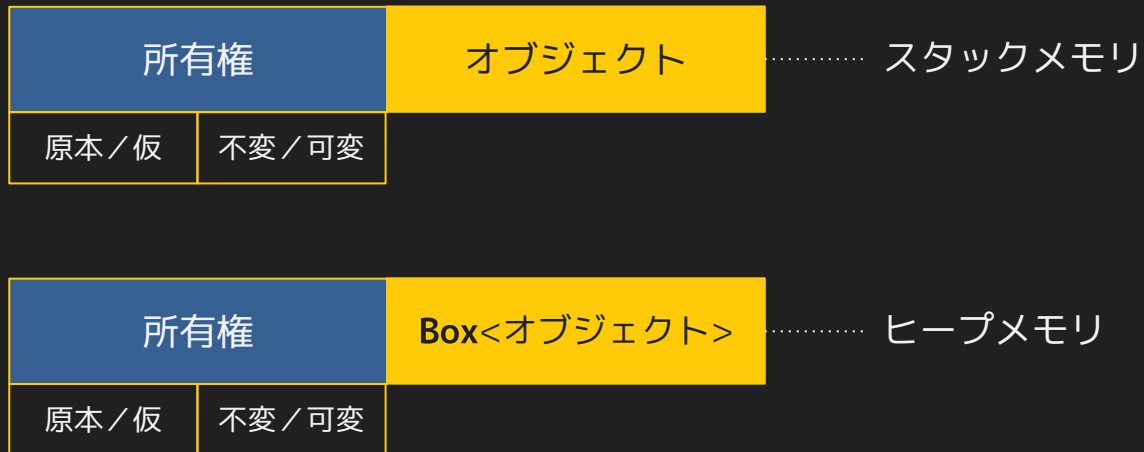
# Box

- この図, 覚えていますか？

所有権		オブジェクト	
原本 / 仮	不変 / 可変		

# Box

- Box を表すと…





# Rc

- **Rc (Reference Count)** とは**参照カウンタ**のことです。原本の所有権を束縛できるのは1つだけです。参照を使えば、仮の所有権を作成することができます。しかし、参照は制約が強いです。同じオブジェクトを複数から束縛することは出来ないのでしょうか。それを可能にするのが参照カウンタで、`Rc<T>` です。`Rc` は原本または仮の所有権を保持することが出来ます。基本的な機能としては参照と変わらず、仮の所有権を作成します。ただし、`&` 演算子ではなく `clone` というメソッドです。

```
use std::rc::Rc;

let a = Rc::new(10);
let b = a.clone();
```

- 通常は、原本の所有権を束縛した変数が破棄されるときは、仮の所有権を持った変数は存在してはいけません。しかし、`Rc` の場合は、原本の所有権を束縛した変数が破棄されたときに、`clone` で作成した仮の所有権を束縛した変数が存在することができます。このとき、オブジェクトは破棄されず、すべての仮の所有権を束縛した変数が破棄されたときにオブジェクトが破棄されます。

- また, この図が出てきました.

所有権		オブジェクト	
原本 / 仮	不変 / 可変		

# Rc

- Rc を表すと…

Rc<所有権>		オブジェクト	
原本 / 仮	不変 / 可変		

# Rc + Box

- 所有権を参照カウントで管理し、オブジェクトをヒープメモリに置くときは次のようになります。

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(Box::new(10));
    let b = a.clone();
    println!("{}", a, b);
}
```

**Rc**<所有権>

**Box**<オブジェクト>

# 内部可変性

- Rust には制限付きで、不変オブジェクトを安全に可変にする方法が用意されています。この実装は内部可変性パターン (**Interior mutability**) と呼ばれるものが使われています。ここでは、詳しく説明しませんが、内部的には `unsafe` で実装されています。この機能を使うには `Cell`, `RefCell` というものを使います。興味がある人は調べてみてください。

# 内部可変性

- またまた、この図が出てきました。

所有権		オブジェクト	
原本 / 仮	不変 / 可変		

# 内部可変性

- `Cell`, `RefCell` を表すと…

所有権		オブジェクト
原本 / 仮 <code>Rc</code>	不変 / 可変 <code>Cell</code> <code>RefCell</code>	



# クロージャ

- **クロージャ**とは、簡単に言うと、変数に束縛できたり、関数の引数として渡すことのできる名前のない関数（**無名関数**）のことです。クロージャはその呼び出し元のスコープにある変数を**キャプチャ**することも出来ます。厳密に言うと、無名関数の中で束縛していない変数のことを**自由変数**と言い、自由変数をまとめた**環境**を無名関数のスコープ内に閉じこめたものをクロージャと呼びます。

# クロージャ

- クロージャは `||` で定義します. 引数があれば `|param1, param2|` のように `||` の間に入れます. その後に `{ }` で本体を記述します. 本体が式 1 つだけなら `{ }` を省略することが出来ます. 次のコードは関数とそれと同じ振る舞いをするクロージャの例です.

```
fn add_one_v1 (x: u32) -> u32 { x + 1 }  
let add_one_v2 = |x: u32| -> u32 { x + 1 };  
let add_one_v3 = |x|           { x + 1 };  
let add_one_v4 = |x|           x + 1 ;
```

# クロージャ

- クロージャはそれぞれ独自 (unique) の型を持っています. クロージャは `Fn`, `FnMut`, `FnOnce` トレイトのどれかのインスタンスです. それぞれ `&self`, `&mut self`, `self` を内部的に引数として受け取っているかどうかの違いがあります.  
また, `Fn` は `FnMut` を, `FnMut` は `FnOnce` を継承しています.

# クロージャ

- 自由変数をまとめた環境をどのように扱うかで、どのトレイトのインスタンスになるかが決まります。まず、すべてのクロージャは必ず `FnOnce` のインスタンスになります。そして、無名関数の中で環境から所有権を移動することがなければ（可変参照は出来る）、`FnMut` のインスタンスになります。さらに、環境を変更しないのであれば、不変参照となるので `Fn` のインスタンスになります。

# クロージャ

- 自由変数が環境にまとめられるとき, 自由変数が束縛しているオブジェクトがコピートレイトのインスタンスであれば, コピーが作成されます. もし, 自由変数をクロージャの中だけで使用することが分かっているならば, 環境にまとめられるときに, コピーではなく所有権を移動することが出来ます. それを行うには, クロージャの前に `move` を指定します.

# Fn と fn

- `Fn` トraitと `fn` というキーワードは別ものです. `fn` は関数定義で使いますが, `fn` は型でもあります. そして, `fn` のことを**関数ポインタ**と言います. `fn` は `Fn` のインスタンスなので, `FnMut`, `FnOnce` のインスタンスでもあります.

# Sized トレイト

- Rust は静的型付け言語です。静的型の特徴の1つとして、型のサイズがコンパイル時に分かることです。しかし、コンパイル時にサイズがわからないこともあります。Rust は型のサイズが分かっているとき、自動でその型を `Sized` トレイトのインスタンスにします。Rust は型が `Sized` トレイトのインスタンスであることを仮定し、それを制約します。つまり、関数の引数などは `Sized` トレイトのインスタンスでなければなりません。ジェネリック型も同じです。それに対して、例えば、`str` 型は実行時にサイズが決まるので、そのような型のことを**動的サイズ型**（**Dynamically sized types: DST**）といいます。

# Sized トレイト

- トレイトは `Sized` トレイトの対象になりません。トレイトはデータ型の分類の仕組みであり、サイズは考慮していないからです。ここで、クロージャに話を戻します。クロージャはトレイトで実装されているので、コンパイル時にサイズがわからないのです。例えば、次のようにクロージャを返す関数はエラーになります。

```
fn returns_closure() -> Fn(i32) -> i32 {  
    |x| x + 1  
}  
// error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static`:  
// `std::marker::Sized` is not satisfied
```



# Sized トレイト

- このような動的サイズ型をどうすれば `Sized` トレイトのインスタンスにすることができるかということですが、参照 (`&`) にするか、`Box` にするかです。例えば、`Box` を使えばクローージャを次のように返すことが出来ます。

```
fn returns_closure() -> Box<Fn(i32) -> i32> {  
    Box::new(|x| x + 1)  
}
```

# 参照を返す関数

- `str` 型は動的サイズ型です. このままでは `Sized` にならないので, 文字列スライスは `&str` 型です. クロージャを返すところでは参照を使わずに `Box` を使いました. 参照を使っても返すことができるのですが, 関数から参照を返すときには**ライフタイム** (**lifetime**) というものを考慮しなければなりません. 個人的にこのライフタイムは余程の理由がない限り扱うべきでないものと思っているので, 本書では詳しく扱いません. なので, 関数から参照を返すのは可能なかぎり避けましょう.

# 動的と静的

- Rust は静的型付け言語にもかかわらず、動的サイズ型もサポートしているのが強みでもあります。これにより静的ディスパッチおよび動的ディスパッチの両方を実現することが出来ます。動的サイズ型は参照や `Box` を使うことで扱えることがわかりました。クロージャはトレイトであり、動的サイズ型であり、参照や `Box` を使うことでオブジェクトとして扱えるようになります。このようなオブジェクトを**トレイトオブジェクト**といいます。

# 動的と静的

- トレイトオブジェクトは、トレイトのメソッドのみ呼び出せることとなります。トレイトオブジェクトは、そのトレイトのインスタンスであればどの型のオブジェクトでも置き換えることができます。このように、トレイトオブジェクトを扱う側は実際のオブジェクトの型を知らなくても、そのメソッドを呼び出せるということ、そしてオブジェクトの型によってメソッドの動作を変えることが出来ることとなります。これらの仕組みを**動的ディスパッチ**といいます。

# 動的と静的

- ジェネリック型や `impl Trait` で指定した型はコンパイル時に型が決まりますので静的です。この `Trait` には任意のトレイト名を指定します。トレイトの型は `Sized` ではないので、参照や `Box` で指定する必要があるのですが、静的である `impl Trait` と区別しやすいように、動的であることを明示する `dyn` が導入され、`dyn Trait` という型を使います。よって、`&dyn Trait` や `&mut dyn Trait`, `Box<dyn Trait>` という形で利用します。

# 動的と静的

- ここでクロージャを返す関数を振り返ってみます。動的と静的を区別するために `dyn` を指定するのですが、これは後から導入されたものなので、省略してもコンパイルは通ります。ただし、警告で付けるように促されるので、実際は次のようになります：

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {  
    Box::new(|x| x + 1)  
}
```

実は `impl` を使うと簡単に静的として扱えます：

```
fn returns_closure() -> impl Fn(i32) -> i32 {  
    |x| x + 1  
}
```

# ライフタイム

- **ライフタイム**というのは参照が有効になるスコープのことです。参照は原本の所有権が存在している限り有効なもので、借用チェックによって厳密にチェックされます。元々、関数に渡すために参照で仮の所有権を渡して破棄してもらう仕組みなのに、それを関数の返り値として返すとはおかしい話です。Rust はパフォーマンスを最優先しているので、仕方ないと思います。返せたほうが便利なきもあるでしょう。Rust の初期版では参照を使っているところは明示的にすべてライフタイムを指定する必要があったようですが、今はかなり緩和されました。

# ライフタイム

- ライフタイムは `&` 演算子の後ろに指定します。慣例的に `a, b, c, ...` と指定します。

```
&i32           // a reference
&'a i32        // a reference with an explicit lifetime
&'a mut i32     // a mutable reference with an explicit lifetime
```

特別なライフタイムの1つに `'static` があります。これはプログラム実行中にずっと存在するライフタイムです。ライフタイムが `'static` なオブジェクトはデータメモリに置かれます。例えば、文字列リテラルは `'static` なライフタイムを持っています。



# ライフタイム

- ライフタイムを指定したコードは本当に見つらいので、なるべくライフタイムを指定するようなコードは書かないほうがいいと思います。

```
impl<'i, 't, 'a, R, P, E: 'i> RuleListParser<'i, 't, 'a, P>
where
    P: QualifiedRuleParser<'i, QualifiedRule = R, Error = E>
        + AtRuleParser<'i, AtRule = R, Error = E>,
{
    pub fn new_for_stylesheet(input: &'a mut Parser<'i, 't>, parser: P) -> Self {
        // ...
    }
}
```

# 並列処理

- 最も基本的な並列処理はスレッドを作成することです。スレッドを作成するには `thread::spawn` を使います。引数にはクロージャを指定します。

```
use std::thread;  
thread::spawn(|| {  
    // thread code  
});
```

`thread::spawn` は `JoinHandle` 型を返します。 `join()` メソッドを呼び出すことで終了を待ちます。 `join()` は `Result` 型を返します。

```
let handle = thread::spawn(|| {  
    // thread code  
});  
handle.join().unwrap();
```

# 並列処理

- クロージャの環境をスレッド間で共有することは通常の方法では出来ません。コピーを作成できるなら、環境にコピーされますが、そうでないなら所有権を移動しなければなりません。その場合は `move` を使います。

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

# 並列処理

- 複数のスレッド間で状態を共有するには**排他制御**が必要です。これを行うために `Mutex` があります。 `lock` メソッドでリソースをロックします。 `lock` メソッドは `LockResult` 型を返します。また, `LockResult` 型は RAII である `MutexGuard` オブジェクトを束縛しているので, 自動でロックを解除します。

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

# 並列処理

- 排他制御を行う `Mutex` は出来ましたが, このオブジェクトをスレッド間で共有しなければなりません. 所有権の共有は `Rc` で出来ませんが, このマルチスレッド版である `Arc` を使います.

```
fn main() {  
    let counter = Arc::new(Mutex::new(0));  
  
    for _ in 0..10 {  
        let counter = Arc::clone(&counter);  
        let handle = thread::spawn(move || {  
            let mut num = counter.lock().unwrap();  
            *num += 1;  
        });  
        ...  
    }
```

# 並列処理

- おや，またこの図が出てきました．

所有権		オブジェクト	
原本／仮	不変／可変		

# 並列処理

- マルチスレッド版の `Rc` である `Arc` , 排他制御を行う `Mutex` の関係を表すと…

`Arc`<所有権>

`Mutex`<オブジェクト>

スレッドセーフ

# Send + Sync

- マルチスレッドでは型の所有権の操作を考慮する必要があります。型の所有権がスレッド間で移動できる場合は、`Send` マーカートレイトのインスタンスになります。`マーカートレイト`とは、メソッドを持たないトレイトのことで、トレイト境界に使うためのものです。`Sized` トレイトもマーカートレイトの1つです。次に、複数のスレッドから安全に参照できる場合は`Sync` マーカートレイトのインスタンスになります。これは参照 `&T` が`Send` ならば、`T` 型は `Sync` であり、参照が別のスレッドに送ることができるという意味になります。



# Send + Sync

- ほとんどのプリミティブ型は `Send+Sync` です。また, `Sync` であるデータ型で構成された型は, それもまた `Sync` です。これは自動的にそれぞれのインスタンスになります。これらを手動で実装するのは安全ではありません。
- マルチスレッドに対応していない `Rc` は `Send` でもなく, `Sync` でもありません（わかりやすく `!Send` や `!Sync` と表記されることもあります）。それに対して, `Arc` は `Send+Sync` です。また, `Arc` が束縛する型もまた `Send+Sync` である必要があります。もし, `Send+Sync` でないなら, `Box` や `Mutex` を利用します。

# RwLock

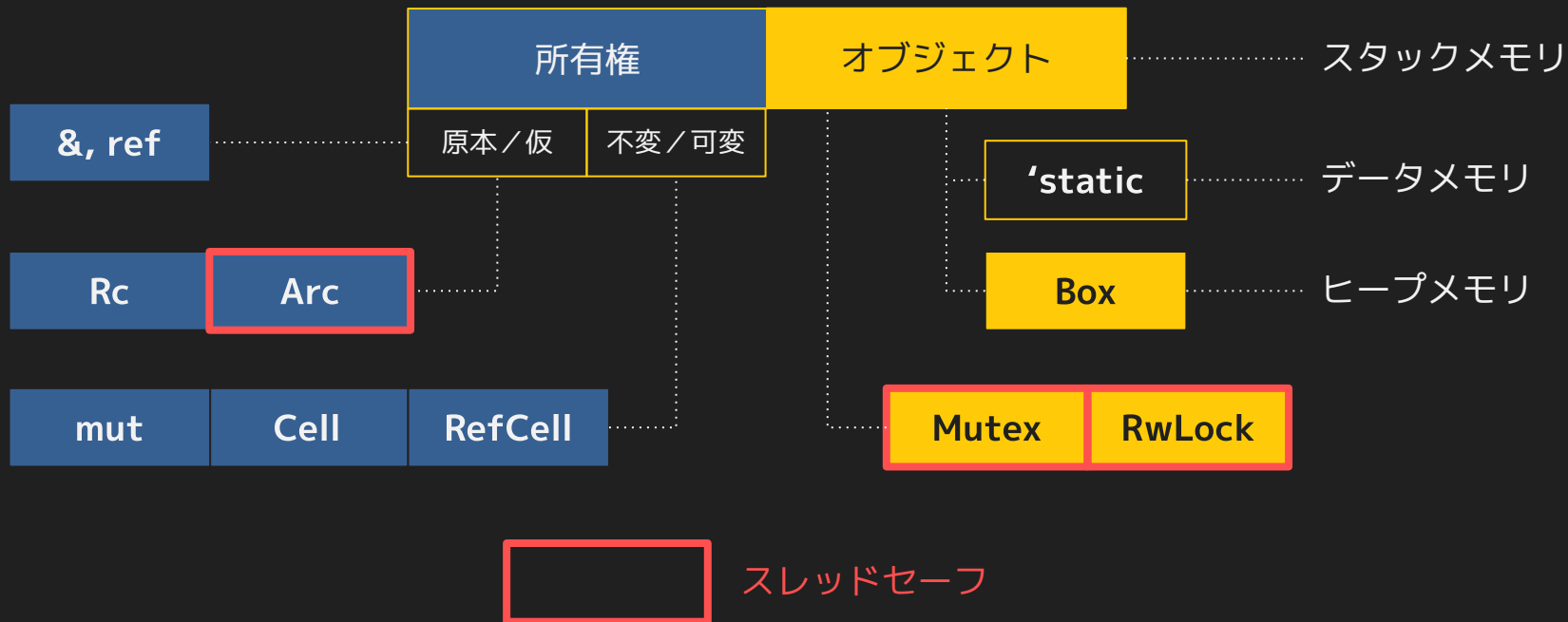
- 排他制御を行うのは `Mutex` 以外に `RwLock` があります. `Mutex` は常に1つのスレッドがリソースの操作をすることが出来ますが, `RwLock` の場合は, 不変参照だけなら複数のスレッドが同時にリソースをロックすることができ, 可変参照のときだけ, 1つのスレッドに制限するものです. 他に**アトミック変数**というの也有ります. これはプリミティブ型のみしか扱えませんが, `Mutex` よりは高速に動作します. 処理速度に問題がなければ基本的に `Mutex` を使うのがいいでしょう. `RwLock` やアトミック変数の詳細は公式リファレンスなどを参照してください.

# 所有権まとめ

- さて、この図に戻ってきました。といっても、今回は最後です。ここまで長かったですね。これまでの内容をまとめてみましょう。

所有権		オブジェクト	
原本 / 仮	不変 / 可変		

# 所有権まとめ



# derive 属性

- これまでいくつかのトレイトを見てきました。実際には多くのトレイトがあり、そして、型は多くのトレイトのインスタンスになっています。トレイトのインスタンスにするには `impl` で実装しなければならず、とても面倒です。そこで、規定の実装をしてくれる機能が用意されています。それが `derive` 属性です。実装したいトレイトを型の定義時に `#[derive(trait, ...)]` という形で指定します。

```
#[derive(Debug, Copy, Clone)]  
pub struct Vec3 {
```

# derive 属性

- 以下は `derive` 属性でよく使われるものです：

<b>Copy</b>	所有権の移動をせずに、複製を作成する
<b>Clone</b>	オブジェクトの複製（ディープコピー）を作成できる
<b>Debug</b>	<code>{:?}</code> で出力できる
<b>Display</b>	<code>{}</code> で出力できる
<b>PartialEq, Eq</b>	<code>==</code> , <code>!=</code> が使える。 <b>Eq</b> はマーカートレイト。
<b>PartialOrd, Ord</b>	<code>&lt;</code> , <code>&gt;</code> , <code>&lt;=</code> , <code>&gt;=</code> が使える。 <b>Ord</b> は順序付けができる。
<b>Send, Sync</b>	スレッド間での移動、参照ができるマーカートレイト。

# From トレイト

- ある型から別の型に変換するときに、便利な `From` トレイトというのがあります. `from` メソッドを対応した型ごとに実装することで、その型から `into` メソッドで変換することが出来ます.

```
#[derive(Debug)]
struct Point { x: f64, y: f64 }

impl From<f64> for Point {
    fn from(input: f64) -> Self {
        Point { x: input, y: input }
    }
}

fn main() {
    let p1 = Point::from(1.0);
    let p2: Point = (1.0).into();
    println!("{:?} {:?}", p1, p2);
}
```

# マクロ

- **マクロ**はメタプログラミングの1つで、コードを展開してくれるものです。特に関数の可変長引数に対応していて、多用します。関数マクロは、関数の最後に **!** 演算子が付いたものです。
- **print!**, **println!** は標準出力に文字列を出力するマクロです。 **eprint!**, **eprintln!** は標準エラーに文字列を出力します。 **dbg!** は式を評価してデバッグ表示してくれます。 **unimplemented!** は未実装を表し, **panic!** を起こします。 **todo!** も同じですが, ニュアンスが異なり, 「まだ未実装」という意味です。
- マクロの機能は多いので, 詳しくは公式ドキュメントなどを参照してください。



# イテレータ

- **イテレータ**は連続したオブジェクトを順番に取り扱うための機能を提供するオブジェクトです。配列やスライス、後で解説する**コレクション**でよく使います。イテレータは `Iterator` トレイトのインスタンスです：

```
pub trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // methods with default implementations elided  
}
```

ここで、`type Item` は**関連型**と呼ばれるもので、インスタンスはこの `Item` 型を定義しなければなりません。

# イテレータ

- イテレータには多くの便利なメソッドが定義されています。いくつか紹介します。まずは `zip` です。これは別のイテレータを受け取って合成し、新しいイテレータを返します。要素はタプルになります。

```
let a1 = [1, 2, 3];  
let a2 = [4, 5, 6];  
let mut iter = a1.iter().zip(a2.iter());
```

# イテレータ

- `map` は各要素に関数を適用します：

```
let a = [1, 2, 3];  
let mut iter = a.iter().map(|x| 2 * x);
```

- `filter` は各要素に対して関数を適用し、`true` を返した要素だけを取り出します：

```
let a = [0i32, 1, 2];  
let mut iter = a.iter().filter(|x| x.is_positive());
```

# イテレータ

- `fold` は状態を持ち、各要素に対して関数を適用して状態を更新し、その状態を返します：

```
let a = [1, 2, 3];  
// the sum of all of the elements of the array  
let sum = a.iter().fold(0, |acc, x| acc + x);
```

- `collect` はイテレータの全要素をコレクションに変換します：

```
let a = [1, 2, 3];  
let doubled: Vec<i32> = a.iter()  
    .map(|&x| x * 2)  
    .collect();
```

# イテレータ

- `enumerate` はインデックスと各要素のペアをタプルで受け取ります：

```
let a = ['a', 'b', 'c'];  
let mut iter = a.iter().enumerate();  
// (0, &'a'), (1, &'b'), (2, &'c')
```

- `inspect` はイテレータの各要素を確認するための `map` です.

`map` では `println!` などが使えないためです：

```
let a = [1, 4, 2, 3];  
let sum = a.iter()  
    .cloned()  
    .inspect(|x| println!("about to filter: {}", x))  
    .fold(0, |sum, i| sum + i);  
println!("{}", sum);
```

# コレクション

- Rust の標準ライブラリには便利な複数のオブジェクトを管理するデータ構造が用意されています。これらを**コレクション**と呼びます。ここでは代表的なコレクションを解説します。

# Vec

- ベクタ `Vec<T>` は伸縮可能な配列です．空のベクタを作成するには `Vec::new` を使います：

```
let v: Vec<i32> = Vec::new();
```

初期値を指定してベクタを作成する場合は `vec!` マクロを使います：

```
let v = vec![1, 2, 3];
```

# Vec

- 各要素を取り出して処理する一般的な方法は `for` 式を使います：

```
let v = vec![100, 32, 57];  
for i in &v {  
    println!("{}", i);  
}
```



# Vec のメソッド

- `Vec<T>` のメソッドの一部です :

```
insert(&mut self, index: usize, element: T)
remove(&mut self, index: usize) -> T
push(&mut self, value: T)
pop(&mut self) -> Option<T>
append(&mut self, other: &mut Vec<T>)
clear(&mut self)
len(&self) -> usize
is_empty(&self) -> bool
first(&self) -> Option<&T>
first_mut(&mut self) -> Option<&mut T>
last(&self) -> Option<&T>
last_mut(&mut self) -> Option<&mut T>
```

# String

- **文字列**を表す `String` もコレクションです. 内部では **UTF-8** でエンコードされたデータです. 文字列型には `OsString`, `OsStr`, `CString`, `CStr`, `String`, `str` などがあります. それぞれ, エンコード方式が違います. `String` と `str` のようにペアになっており, `str` はスライスです.

# String

- 文字列の生成は `String::new` です：

```
let mut s = String::new();
```

文字列以外の型から、文字列に変換するには `to_string` メソッドが便利です。これは `Display` トレイトのインスタンスなら自動で実装してくれます。

```
let i = 5;  
let five = i.to_string();
```

# String

- 文字列リテラルからの作成は `String::from` を使います：

```
let five = String::from("5");
```

文字列から数値型に変換するには `parse` を使います。これは `Result` 型を返します：

```
let a_string = String::from("5");  
let b = a_string.parse::<i32>()?;
```

# String

- 書式付きで文字列を作成するには `format!` マクロを使います：

```
let s1 = String::from("tic");  
let s2 = String::from("tac");  
let s3 = String::from("toe");  
let s = format!("{}", s1, s2, s3);
```

# String メソッド

- String のメソッドの一部です：

```
push_str(&mut self, string: &str)
push(&mut self, ch: char)
pop(&mut self) -> Option<char>
as_bytes(&self) -> &[u8]
truncate(&mut self, new_len: usize)
insert(&mut self, idx: usize, ch: char)
insert_str(&mut self, idx: usize, string: &str)
remove(&mut self, idx: usize) -> char
len(&self) -> usize
is_empty(&self) -> bool
clar(&mut self)
chars(&self) -> Chars<'_>
bytes(&self) -> Bytes<'_>
starts_with<'a,P>(&'a self, pat: P) -> bool
ends_with<'a,P>(&'a self, pat: P) -> bool
find<'a, P>(&'a self, pat: P) -> Option<usize>
rfind<'a, P>(&'a self, pat: P) -> Option<usize>
trim(&self) -> &str
```

# HashMap

- `HashMap<K,V>` は**連想配列**です。オブジェクトにキーを結びつけて管理します。空の連想配列を作成するには `HashMap::new` を使い、新しい要素を挿入する場合は `insert` を使います：

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

# HashMap

- キーに対応した要素を取得するには `get` を使います：

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```



# HashMap

- 各要素を取り出す場合は `for` 式で、キーとオブジェクトを分解束縛します：

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

# HashMap

- キーがまだ存在していないときに挿入する場合は `entry` と `or_insert` を使います：

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);
println!("{:?}", scores);
```

# HashMap

- 2つのベクタから連想配列を作成するにはイテレータを使って, `zip`, `collect` を使う方法があります :

```
use std::collections::HashMap;

let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let mut scores: HashMap<_, _> =
    teams.into_iter().zip(initial_scores.into_iter()).collect();
```

# HashMap

- `HashMap<K,V>` のメソッドの一部です：

```
keys(&self) -> Keys<'_, K, V>
values(&self) -> Values<'_, K, V>
values_mut(&mut self) -> ValuesMut<'_, K, V>
iter(&self) -> Iter<'_, K, V>
iter_mut(&mut self) -> IterMut<'_, K, V>
len(&self) -> usize
clear(&mut self)
entry(&mut self, key: K) -> Entry<'_, K, V>
contains_key<Q: ?Sized>(&self, k: &Q) -> bool
insert(&mut self, k: K, v: V) -> Option<V>
remove<Q: ?Sized>(&mut self, k: &Q) -> Option<V>
```

# Cargo

- **Cargo** はパッケージ管理ツールです。 **パッケージ**とは1つ以上のクレートを含んだもののことです。そして、 **クレート**とは Rust プログラムをビルドしたもので、 **バイナリクレート**（実行可能ファイル）と **ライブラリクレート**の2つがあります。パッケージは複数のバイナリクレートを含めることができますが、ライブラリクレートは1つまでしか含めることが出来ません。 **モジュール**はクレートの中でグループ化されたコードのことで、読みやすさと再利用性を高めるためのものです。また、 **プライバシー**を設定することができ、内部の実装を利用できなくすることも出来ます。

# Cargo

- `init` コマンドを使うと、実行したフォルダ内に、単一のバイナリクレートを含んだパッケージの作成環境が作られます。

```
cargo init
```

実行すると次のような構成になります。

```
(workspace)
├── src
│   └── main.rs
├── cargo.toml
└── .gitignore
```

# Cargo

- `cargo.toml` は構成ファイルです.
- `src/main.rs` が**クレートルート**です. このファイルがモジュール構造の起点となります.
- `build` コマンドを実行するとビルドが始まります. Cargo は必要な外部パッケージなどを自動でダウンロードしてビルドします. 依存する外部パッケージは `cargo.toml` に記述し, 実際にダウンロードしたパッケージのバージョンを `cargo.lock` ファイルに書き込みます.

```
cargo build
```

# Cargo

- `run` コマンドを実行すると、作成した実行可能ファイルを起動します。ビルドが必要な場合は自動的にビルドしてくれます。

```
cargo run
```

- `check` コマンドはコンパイルチェックをします。 `build` コマンドとは違ってリンクを行わないので、コンパイルが通るかのチェックはこちらのほうが早いです。

```
cargo check
```



# モジュール

- モジュールの定義は `mod` を使います. モジュールの本体は `{ }` で囲みます. モジュールの中にモジュールを定義することも出来ます.

```
mod Module1 {  
    mod Module11 {  
        mod Module111 {  
            fn hoge() {}  
        }  
    }  
  
    mod Module12 {  
        fn foo() {}  
        fn bar() {}  
    }  
}
```

# モジュールの公開

- モジュールは同じモジュール内に対してだけ公開された状態になります。そこで、外部のモジュールに対しても公開するには `pub` を使います。  
`pub` はモジュール、関数、構造体などに1つずつ設定することが出来ます

```
pub mod Module1 {  
    pub mod Module11 {  
        pub mod Module111 {  
            pub fn hoge() {}  
        }  
    }  
}
```

# パス

- モジュールを利用するには**パス**が必要です。モジュールの起点はクレーンルートで、パスは `crate` になります。前のコードが `src/main.rs` に含まれていた場合、それぞれのパスは次のようになります。

```
crate
├── Module1
│   ├── Module11
│   │   └── Module111
│   │       └── hoge
│   └── Module12
│       ├── foo
│       └── bar
```

# パス

- パスの指定には2種類あります。絶対パスと相対パスです。絶対パスはクレートルートを表す `crate` , または外部パッケージおよび標準ライブラリの場合はパッケージ名から指定することが出来ます。相対パスの場合は `self` , または `super` を使います。 `self` は現在のモジュールから, `super` は親のモジュールからの指定になります。パスの区切りは `::` を使います。ちなみに `self::` は省略出来ます。

```
crate::Module1::Module11::Module111::hoge
```

# モジュールの利用

- モジュールを利用するには `use` を使ってパスを指定します：

```
use crate::Module1::Module11::Module111::hoge;  
use std::fmt::Result;
```

- `use` でモジュールのデータ型を指定した場合は `as` で別名をつけることができます：

```
use std::io::Result as IoResult;
```

# モジュールの利用

- `use` で指定するパスにおいて、あるモジュールから別々のパスを指定する場合、それぞれのパスを `use` で指定すると必要な行が増えてしまいます。そこで、パスのリストを記述することが出来ます：

```
use crate::Module1::{Module11::Module111, Module12};
```

- また、あるモジュール以下をすべて現在のスコープで利用する場合はグローバル（`*`）を指定することができます：

```
use crate::Module1::*;
```

# モジュールツリー

- クレートそのものがモジュールであり, クレートルートは `src/main.rs` ファイルで, パスは `crate` でした. Rust のモジュールシステムはクレートルート以下のフォルダとファイルもまたモジュールと見なします. これによって別々のファイルに実装を分けることが出来るわけです. このフォルダとファイルによるモジュールの作り方が2種類あります. 先に一般的な方法を説明します.

# モジュールツリー

- `mod` では `{ }` で囲む他に、指定した名前と同じファイルを同じフォルダ内から検索して、その中身を挿入することが出来ます。例えば、`src/hoge.rs` というファイルがあるとします。`src/main.rs` から `mod hoge;` とすれば `src/hoge.rs` の中身を `src/main.rs` ファイルに挿入します。ここで `src/hoge.rs` の中身が次のようになっているとします。

```
pub fn hoge() {}
```

この場合、`src/main.rs` からは `crate::hoge::hoge()` で呼び出すことが出来ます。



# モジュールツリー

- 次に右図のようなフォルダ構成を考えます．このような構成にすると，`src/main.rs` から `mod module1;` とすることで，`src/module1/mod.rs` ファイルが読み込まれます．そのファイルの中身は `pub mod foo;` とします．これで，`src/main.rs` から `src/module1/foo.rs` のモジュールを利用することが出来ます．同じように `pub mod bar;` を `mod.rs` に追加すれば `bar.rs` モジュールも利用できるようになります．このようにモジュールの階層を作ってプログラムを構築していきます．これを**モジュールツリー**といいます．

```
(workspace)
├── src
│   ├── main.rs
│   └── module1
│       ├── mod.rs
│       ├── foo.rs
│       └── bar.rs
├── cargo.toml
└── .gitignore
```

# モジュールツリー

(workspace)

src

main.rs

module1

mod.rs

foo.rs

bar.rs

cargo.toml

.gitignore

`mod module1;`

`pub mod foo;`  
`pub mod bar;`

# モジュールツリー

- もう1つのモジュールツリーの作り方ですが、右図を見てください。今度は `mod.rs` ファイルの代わりに、フォルダと同じファイル `module1.rs` が存在しています。このファイルの中身は `mod.rs` と同じになります。ファイルの構成が違いますが、モジュールツリーは同じなので、コードに変更はありません。

```
(workspace)
├── src
│   ├── main.rs
│   ├── module1.rs
│   └── module1
│       ├── foo.rs
│       └── bar.rs
├── cargo.toml
└── .gitignore
```

# モジュールツリー

(workspace)

src

main.rs

module1.rs

module1

foo.rs

bar.rs

cargo.toml

.gitignore

`mod module1;`

`pub mod foo;`  
`pub mod bar;`

# モジュールの再公開

- `pub use` を使うことで、外部モジュールからでも、指定したパスで利用できるようになります。これを**再公開**といいます。

```
pub use crate::module1;
```

# 外部パッケージの利用

- Cargo.toml の dependencies セクションに使用したいパッケージを指定します。例えば rand パッケージを使う場合は次のようにします：

```
[dependencies]
rand = "0.8.0"
```

現在のパッケージ構成を確認したい場合は以下のコマンドを使います：

```
cargo tree
```

# ユニットテスト

- Rust にはテストコードを記述する機能が用意されています。テストを実行するには以下のコマンドを実行します：

```
cargo test
```

実行するユニットテストの関数には `test` 属性を付けます：

```
#[test]
fn it_works() {
    assert_eq!(2 + 2, 4);
}
```

# ユニットテスト

- テスト時のみ（ `cargo test` ）ビルドするモジュールを作ることが出来ます。それにはモジュールの前に `cfg(test)` 属性を付けます。

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```



# ユニットテスト

- テストに便利なマクロがいくつか用意されています。 `assert!` マクロは引数が `true` かどうかをテストします。また、`==` や `!=` 演算子を使う代わりに `assert_eq!`, `assert_ne!` マクロが用意されています。

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert!(2 == 2);
        assert_eq!(2 + 2, 4);
        assert_ne!(2 + 2, 5);
    }
}
```

# ユニットテスト

- `assert!` マクロや, `unwrap` などは `panic!` を呼び出す場合があります. 場合によっては `panic!` が呼び出されることを期待したテストコードを記述したい場合があります. しかし, テストは通常 `panic!` を起こすと失敗になります. そこで, `should_panic` 属性を付けることで, `panic!` を起こすことがテストの目的であることを明示します:

```
#[test]
#[should_panic]
fn it_works() {
    panic!();
}
```

# 未使用の変数

- プログラムの中に未使用の変数があればコンパイル時に警告が出ます。その場合は変数の前にアンダースコア ( `_` ) を付けることで、警告を抑制できます。

# 分解束縛

- 分解束縛において、多くのフィールドがあるときに、必要なものだけ束縛して、残りは無視したいことがあるかもしれません。そのときは、`..` を使用します：

```
struct Point {  
    x: i32,  
    y: i32,  
    z: i32,  
}  
  
let origin = Point { x: 0, y: 0, z: 0 };  
  
match origin {  
    Point { x, .. } => println!("x is {}", x),  
}
```

# ドキュメント

- オフラインで Rust のドキュメントを参照するには次のコマンドを使います

```
rustup doc
```

現在のワークスペースに関するドキュメントを参照するには次のコマンドを使います

```
cargo doc --open
```

# 最後に

- いかがだったでしょうか。思っていた以上に長くなってしまいましたが、それでも全部説明しきれていないのが本音です。ただ、一番解説したかった所有権のところはしっかり書いたつもりです。ここで、説明されていないところ、例えばドキュメントコメント、`rustfmt`、`clippy`などは公式ドキュメントなどを参照してください。とはいえ、正直あまり公式ドキュメントはオススメしません。そもそもこの入門を書いたのも、公式ドキュメントがとてもわかりづらいと思ったからです。

# 最後に

- 私も Rust を本気で勉強してまだ2～3ヶ月ぐらいだと思います。Rust 入門みたいな本は一冊も購入していないし読んでいません。基本的に公式ドキュメントや Tour of Rust, ネットで公開されている情報だけです。なので, 大いに勘違いしているかもしれません。なにか間違っているところがあればご連絡していただけると嬉しいです。少しでも参考になれば幸いです。
- twitter: [@mebiusbox2](https://twitter.com/mebiusbox2)

# 参考

- [The Rust Programming Language](#)
- [The Rust Programming Language 日本語版](#)
- [Tour of Rust](#)
- [Rust by Example](#)
- [Rust Playground](#)
- [Rust for a Pythonista #2: Building a Rust crate for CSS inlining](#)
- [Rustに影響を与えた言語たち](#)
- [RustでOption値やResult値を上手に扱う](#)
- [最速で知る！プログラミング言語Rustの基本機能とメモリ管理【第二言語としてのRust】](#)
- [ウォークスルー Haskell](#)