5～10目のように関数内で任意のブロックを作成できます。ブロック内では整数型の変数totalを宣言し、forループで利用しているカウンタiの値を加算しています。加算が終わったら結果をブロックの外で出力しています。

変数totalのスコープは5～10行目の範囲になるため、スコープ外となりコンパイルエラーになります。

```
error[E0425]: cannot find value `total` in this scope
```

変数totalをブロックの外で宣言するか、結果の出力をブロック内で実行するかのいずれを考えることができます。Rustのコンパイルエラーメッセージの先頭は「error[Exxxx]」になり、必ずエラー番号が付いています。エラー番号に対応するエラーの説明やサンプルコードが公開されています。

```
・https://doc.rust-lang.org/error_codes/error-index.html
```

## 3-2 静的変数

他言語と同様にstaticキーワードを利用して静的な変数を宣言できます。目的に応じて関数やメソッドの内外で宣言可能です。静的変数の利用目的は複数の処理からの値の共有がほとんどです。

Rustの変数は、デフォルトでイミュータブル（変更不可）になるため、複数スレッドからのアクセスがあった場合でも問題ありませんが、静的変数に限らず、ミュータブルな変数の場合は競合を防ぐ排他制御が必要になります。この節では静的変数の宣言とミュータブルな変数の競合を防ぐRustの機能について解説します。

### 3-2-1 静的変数の宣言

静的変数宣言にはletキーワードではなく、staticキーワードを利用します。mutキーワードを付加すればミュータブルは静的変数を宣言できます。変数宣言には必ず型名を指定する必要があります。名称は大文字で定義しないとワーニングメッセージが出力されます（**リスト3.6**）。

```
static 変数名: 型名 = 初期値;
static mut 変数名: 型名 = 初期値;
```

**リスト3.6 静的変数の宣言と利用（static_val.rs）**

```
1:  ///  ## 3-2.静的変数
2:  ///  ### リスト3-6 消費税率を表す静的変数
3:  static TAX_RATE:f32 = 0.10;
4:  ///  ## 3-2.静的変数
5:  ///  ### リスト3-6 静的変数の宣言と利用
6:  ///  #### 引数:price 単価
7:  ///  #### 戻り値:消費税込みの金額
8:  #[allow(dead_code)]
9:  pub fn calc_amount(price: i32) -> i32 {
10:     let fprice = price as f32; // i32型の値をf32型に変換する
11:     let result = fprice + fprice * TAX_RATE; // 消費税込みの金額を計算する
12:     result as i32 // f32型からi32型に変換して結果を返す
13: }
```

```
単価:100の税込み金額=110
```

3行目で消費税率を表す静的変数を宣言し、それを利用して税込金額の計算結果を返すcalc_amount()関数で利用しています。引数でi32型の単価を受け取っていますが、消費税率f32（浮動小数点）型になっています。型が異なるので、演算の際にエラーになってしまいます。

asキーワードを利用して型のキャストができます。12行目では税込み金額をi32型に変換して返しています。Rustではreturnキーワードを利用せず、セミコロンのない文で戻り値を返すことができます。

### 3-2-2 unsafeキーワード

マルチスレッドでミュータブルな静的変数を利用する場合は、スレッドの排他制御をしないと正しい値が保証できなくなってしまいます。

まずは、単純にミュータブルな静的変数を宣言、利用しようとした場合にどうなるかを見ていきましょう（**リスト3.7**）。

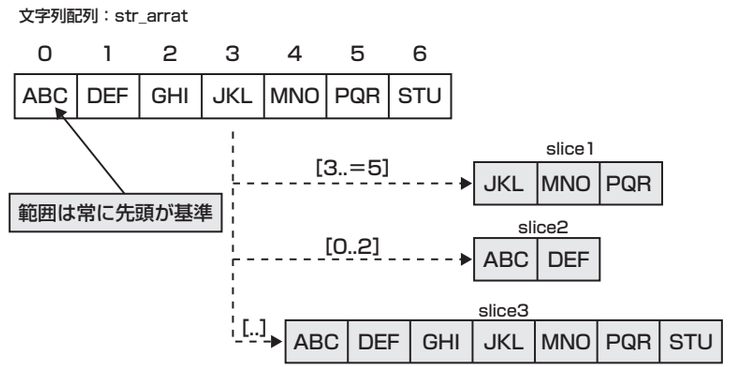
**リスト3.7 ミュータブルな静的変数の宣言と利用（static_val.rs）**

```
1:  ///  ## 3-2.静的変数
2:  ///  ### リスト3-7 ミュータブルな静的変数の利用
3:  static mut TOTAL_VALUE:i32 = 0;
4:  ///  ## 3-2.静的変数
5:  ///  ### リスト3-7 ミュータブルな静的変数の利用
6:  pub fn calc_total(value: i32) {
7:      TOTAL_VALUE += value;
8:      println!("TOTAL_VALUE = {}" , TOTAL_VALUE);
9:  }
```

```
11:     println!("slice2 = {:?}" , slice2);
12:     println!("slice3 = {:?}" , slice3);
13: }
```

```
slice1 = ["JKL", "MNO", "PQR"]
slice2 = ["ABC", "DEF"]
slice3 = ["ABC", "DEF", "GHI", "JKL", "MNO", "PQR", "STU"]
```

図4.1　スライスの範囲




## 4-7-2　Range構造体

Range構造体は**リスト4.21**のように範囲を表す情報を保持します。startフィールドは開始（範囲の下限）、endフィールドは終了（範囲の上限）を表します。範囲start..endには、「start <= x <end」のすべての値が含まれ、「start >= end」の場合は空になります。構造体は範囲を表現するものであり、直に利用することもできれば糖衣構文の利用もできます。

リスト4.21　std::ops::Range構造体

```
1: pub struct Range<Idx> {
2:     pub start: Idx,
3:     pub end: Idx,
4: }
```

構造体は型パラメータでIdx型を指定しています。このためstartとendのデータ型にはIdx型が割り当てられています。Idx(idx::Idex)型は構造体で、キーで値をマッピングするインデックスを表現しています（**リスト4.22**）。

リスト4.22　Range構造体の利用（slice_type.rs）

```
1:  /// ## 4-7 スライス型
2:  /// ### リスト4-22 Range構造体の利用
3:  #[allow(dead_code)]
4:  pub fn range() {
5:      // スライスを取得する文字列配列
6:      let int_array = [1 , 2 , 3 , 4 , 5 , 6 , 7];
7:      // Range構造体を生成する
8:      let range = std::ops::Range{start:1 , end:3};
9:      // スライスを取得する
10:     let slice = &int_array[range];
11:     println!("slice = {:?}" , slice);
12: }
```

```
slice = [2, 3]
```

## 4-7-2　マルチバイト文字列の利用

Rustでは日本語などのマルチバイト文字は、1文字につき3バイト消費します。このため、スライスの範囲指定では、0、3、6、9のように0と3の倍数を利用して範囲指定します。範囲指定にミスがあるとパニックになります。

不正な範囲指定の例と正しい範囲指定の例を**リスト4.23**、**リスト4.24**に示します（**図4.2**）。

リスト4.23　不正な範囲指定（slice_type.rs）

```
1:  /// ## 4-7 スライス型
2:  /// ### リスト4-23 マルチバイトの利用
3:  #[allow(dead_code)]
4:  pub fn multibyte_slice() {
5:      let comany_name = String::from("株式会社フルネス");
6:      // 不正な範囲指定
7:       let slice = &comany_name[1..3];
8:      println!("参照範囲={:?} , 大きさ={}" , slice , slice.len());
9:  }
```

```
thread 'main' panicked at 'byte index 1 is not a char boundary; it is inside '株' (bytes 0..3)
 of `株式会社フルネス`', src\slice_type.rs:40:18                                    実際は一行
…
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
error: process didn't exit successfully: `target\debug\chapter04.exe` (exit code: 101)
```

```
11:             "変数numの値は1でも2でもありません。"
12:         };
13:         println!("{}" , result);
14:     }
```

```
変数numの値は1でも2でもありません。
```

## 5-2 パターンマッチング

パターンマッチングは、C/C++やJavaのswitch～case文と似た制御文で、式の値がパターンに一致するかしないかを判定します。Rustでは関数やメソッドの戻り値としてResult<T,E>、Option<T>型がよく用いられます。関数やメソッドから返されるこれらの型をmatch式で評価します。本節では単純な値の評価を紹介しますが、以降の関数やメソッドをテーマにした章で実務的な利用方法についてサンプルコードとともに解説します。

### 5-2-1 match式

match式は複数の値で評価し、評価する値と一致するパターンに実装された処理が実行されます。

式の評価は「パターン => 処理ブロック」をカンマ区切りで指定します。「パターン => 処理ブロック」を「アーム」と言います。

match式は、アームのパターンを順番に評価していき、一致した処理ブロックのみを実行します。どのアームにも一致しないパターンは「ワイルドカード(_)」を使って定義します。matchキーワードの次に記述する式の部分は、値、式、関数やメソッドの呼び出しを記述できます(**リスト5.4**)。値は数値形式だけでなく文字や文字列も利用できます(**リスト5.5**)。

実行する処理が1つの場合は、ブロックとセミコロンを省略できます。

```
match 式 {
    パターン1 => { 処理1; },
    パターン2 => { 処理2; },
    パターン3 => { 処理3; },
    ...
    _   => { どのパターンにも一致しなかった場合の処理; }
}
```





リスト5.4 整数値を評価するmatch式(branch_match.rs)

```
1:  /// ## 5-2.パターンマッチング
2:  /// ### リスト5.4 match式
3:  #[allow(dead_code)]
4:  pub fn branch_1(){
5:      let x = 10;
6:      match  x { // 単純なmatch式
7:          1 => println!("値は1"),
8:          2 => println!("値は2"),
9:          _ => println!("値は不正です。")
10:     }
11:     match  x { // 複数の処理をするmacth式
12:         1 => {
13:             let y = 100;
14:             println!("y = {}" , y);
15:         },
16:         2 => {
17:             let y = 200;
18:             println!("y = {}" , y);
19:         },
20:         _ => {
21:             let y = 300;
22:             println!("y = {}" , y);
23:         }
24:     }
25: }
```

```
値は不正です。
y = 300
```

リスト5.5 文字列を評価するmatch式(branch_match.rs)

```
1:  /// ## 5-2.パターンマッチング
2:  /// ### リスト5.5 match式
3:  #[allow(dead_code)]
4:  pub fn branch_2(){
5:      let x = "山田太郎";
6:      match x {
7:          "山田太郎" => println!("山田太郎です。"),
8:          "鈴木花子" => println!("鈴木花子です。"),
9:          _ => println!("誰?")
10:     }
11: }
```

```
山田太郎です。
```

```
x + y = 30
```

少し冗長な処理内容になっていますが、typeキーワードを利用し、add()関数が代入可能なCalc型を宣言しています。typeキーワードの後に型名を記述し、代入演算子の後に型の構造を宣言しています。型名はガイドラインの命名規則に従いCamelCaseにしています。関数を代入するため、fnキーワードとカッコの中に引数の型、次に戻り値を指定します。

6～8行目にあるuse_calc_type()関数は最初の引数funcの型名をCalc型にすることで、add()関数を引数で受け取り、引数名で関数を実行しています。

12～16行目のuse_finction_2()関数では、Calc型変数calcにadd()関数を代入し、use_calc_type()関数に加算対象の値とともに渡して実行しています。

### 7-2-3　関数型を返す

引数で関数を渡すことが可能ですから、当然戻り値で関数を返すこともできます（**リスト7.10**）。

**リスト7.10　関数型を返す（function_type.rs）**

```
1:  /// ## 7-2 関数型
2:  /// ### リスト7.10 関数型を返す
3:  /// ### 戻り値 Cald
4:  #[allow(dead_code)]
5:  fn return_calc_type() -> Calc {
6:      add // add()関数をリターンする
7:  }
8:  /// ## 7-2.関数型
9:  /// ### リスト7.10 関数型を返す
10: #[allow(dead_code)]
11: pub fn use_function_3() {
12:     let calc = return_calc_type(); // 関数型Calcを受け取る
13:     let r = calc(10 ,20);// 受け取った関数を実行する
14:     println!("10 + 20 = {}" , &r);
15: }
```

```
x + y = 30
```

5～7行目のreturn_calc_type()関数の戻り値はCalc型になってきます。処理ブロックではadd()関数をaddのみを記述してリターンしています。

11～15行目のuse_function_3()関数ではreturn_calc_type()関数からCalc型の関数を受け取り実行しています。





## 7-3　ジェネリクスとトレイト境界

関数を作成していると、型は異なるが似たような処理を多く実装しているケースが出てきます。関数の引数や戻り値を実行するタイミングで指定することで、別々に実装していた関数を1つにまとめることが可能になる場合があります。他言語の多くが型パラメータをサポートし、実行時に型を決めていろいろな型を処理できるようにしています。

**6章**のBox<T>型の解説でも述べましたが、Rustでは型パラメータのことを「ジェネリクス」といい、任意に選択できる型を「ジェネリック型」と言います。任意の型を決めて利用できたとしても、処理内容にそぐわない型を指定しても意味がありません。例えば、前節では加算機能をサンプルに関数型の利用を説明してきましたが、ジェリック型に文字列やその他の加算に無関係な型を指定しても処理ができませんし使えません。

Rustは型に対して、大変細かくトレイトを利用して特性を付け加えています。std::ops::Addトレイトを実装していることで、加算機能が利用できます。加算処理を実現したい場合は加算できる特性を持った型をジェリック型に指定できなければ意味がありません。このような場合、引数や戻り値で指定できる型は特定の特性を持った型のみを受け付けるように制約を設けることができ、これを「トレイト境界」と言います。

この節では加算と減算機能を例にジェネリクスとトレイト境界を利用した関数について解説します。トレイトは以降で解説しますが、最初にAddトレイトとSubトレイトを解説をしてから関数の実装方法と利用方法について解説します。

### 7-3-1　Add／Subトレイト

Add、Subトレイトは以下のように実装されています（**リスト7.11**、**リスト7.12**）。

**リスト7.11　std::ops::Addトレイトの実装**

```
1:  pub trait Add<Rhs = Self> {
2:      type Output;
3:      fn add(self, rhs: Rhs) -> Self::Output;
4:  }
```

**リスト7.12　std::ops::Subトレイトの実装**

```
1:  pub trait Sub<Rhs = Self> {
2:      type Output;
3:      fn sub(self, rhs: Rhs) -> Self::Output;
4:  }
```

2つのトレイトとも2行目に「type Output」という記述があります。そして、宣言されたメソッ

# 10章 トレイト

**トレイトは構造体や列挙型など任意の型に特性を持たせるために利用します。標準ライブラリのデータ型は多くのトレイトを実装することで、型特有の特性を持っています。Rustを使ったプログラミングでは標準ライブラリが提供するトレイトを利用するだけでなく、ある特性を任意のトレイトで表現し、それを実装していくのが一般的です。本章ではトレイトの宣言と実装方法を解説するとともに、ジェネリックトレイトや関連型トレイトといわれる汎用的に利用可能な振る舞いを実装するためのトレイト、オブジェクト指向プログラミング言語のポリモーフィズムのようにトレイトを実装した型を抽象的に扱う方法について解説します。**

## 10-1 トレイトの基本

トレイトは、様々な型に特定の特性と振る舞いを追加するために用いられます。特性だけを表すマーカートレイトも含め、標準ライブラリは多数のトレイトを提供し、Rustの世界では無くてはならない存在です。

トレイトは提供されるものを利用するだけでなく、任意の目的で宣言して利用できます。この節ではトレイトの基本的な宣言と型への実装について解説します。

### 10-1-1 トレイトの宣言

トレイトはtraitキーワードを利用して宣言します。構造体や列挙型のようにブロック中に型関連関数、メソッドを宣言します。トレイトも名称の先頭は大文字にして実装します。型関連関数やメソッドの宣言の最後にはセミコロンを付加します。宣言するメソッドや型関連関数には予めデフォルトの実装をすることもできます。

**リスト10.1**は簡単な計算機能を表すトレイトの宣言例です。

```
trait トレイト名 {
    // 宣言のみ
    fn メソッド名(&self または&mut self,[引数名:型名,..] )[-> 型名];
    fn 型関連関数名([引数名:型名,..]) [-> 型名];
    ...
    // デフォルト実装
    fn メソッド名(&self または&mut self,[引数名:型名,..] )[-> 型名]{
        実装
    }
    fn 型関連関数名([引数名:型名,..]) [-> 型名]{
        実装
    }
}
```

リスト10.1　計算機能を表現したトレイト（basic.rs）

```
1:  use anyhow::Result;
2:  /// ## 10-1.トレイトの基本
3:  /// ### リスト10.1 計算機能を表現したトレイト
4:  pub trait Calculator {
5:      /// 計算処理メソッド
6:      fn calc(&self) -> Result<u64>;
7:  }
```

**リスト10.1**は計算機能を表していますが、どのような計算をするのかは決まっていません。calc()メソッドには実装が無いため、宣言の最後はセミコロンで終わっています。

### 10-1-2 todo!()マクロ

**リスト10.1**のメソッドには実装がありませんでした。**リスト10.2**のうようにtodo!()マクロを利用して実装部分を作っておくこともできます。

リスト10.2　todo!()マクロ（basic.rs）

```
1:  use anyhow::Result;
2:  /// ## 10-1.トレイトの基本
3:  /// ### リスト10.2 todo!()マクロ
4:  pub trait Calculator {
5:      /// 計算処理メソッド
```

# 11章 エラー

**これまでエラー及びその対処についてあまり触れてきませんでした。Rustは大変強力な型推論機能があるためサンプルコードでも型推論を使い、関数やメソッドから戻されてくる値の型については解説レベルにととどまっていました。本格的なプログラムを作成する場合、エラーへの対応は品質という観点から大変重要になります。Rustにはエラーの仕組みとしてパニックとエラーがあり、実装内容によって使い分けが必要になります。エラーについては他言語と同様に、標準ライブラリに多数のエラー型が提供されています。この章ではRustのエラー機能の基本とパニックについて解説するとともに、外部クレートを利用したエラー型の作成と利用について解説します。**

## 11-1 エラー型の基本

標準ライブラリにはプログラム作成のための様々な機能要素と、それらがエラーになったことを通知するためのエラー型が多数提供されています。また、エラーの種類によってはエラー型だけでなく、エラーのより具体的な内容を表す列挙型も提供さています。

この節では、標準ライブラリが提供するエラーの基本的な知識について解説します。なお、解説する内容は外部クレートでも基本的に同じです。

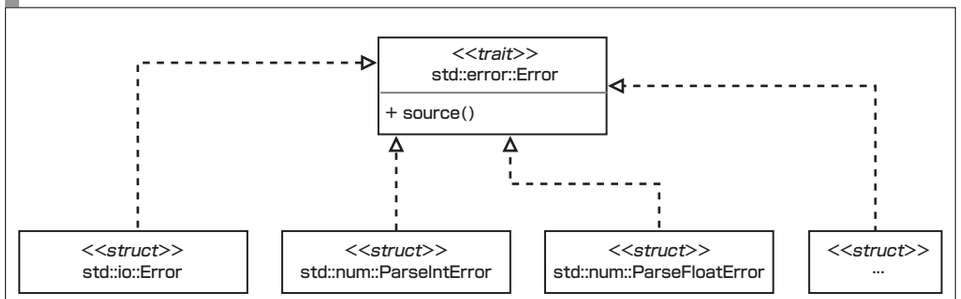
### 11-1-1 エラー型の基本構成

エラー型の基本となるのが、std::error::Errorトレイトになります。様々な機能が返すエラー型はこのトレイトを実装しています。source()メソッドは別のエラー型を保持する機能で、Option型で保持しているエラーを返す機能実装のために利用されます。

**図11.1**には入出力エラーを表すstd::io::Error型と、型変換機能のparse()メソッドで返されるstd::num::ParseIntErrorとParseFloatError型を記述していますが、その他にも多くのエラー型が提供されています。ここではあまりに多いので割愛しています。

実装例を**リスト11.1**に示します。

図11.1 エラー型の基本構成




リスト11.1 parse()メソッドのエラー型（basic.rs）

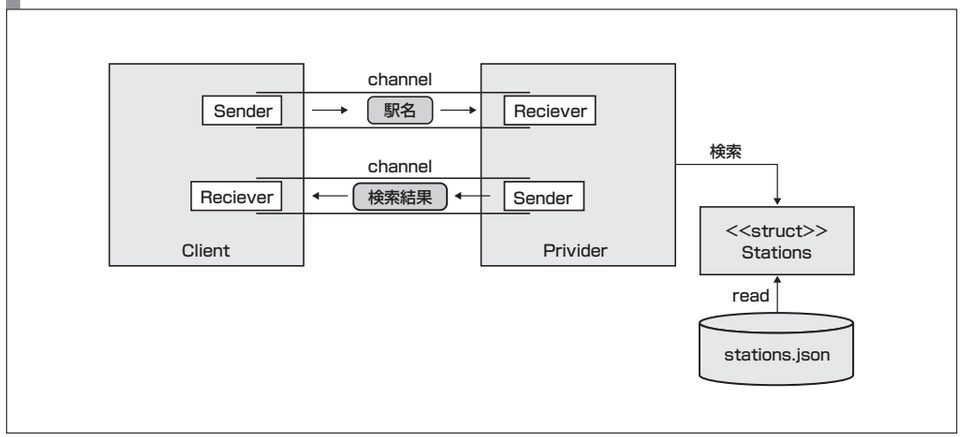
```
/// ## 11-1.エラー型の基本
/// ### リスト11.1 変換種別を表す列挙型
#[derive(Debug , PartialEq, Eq)]
pub enum ValueConversion {
    Int , // i32整数に変換する
    Float // f32浮動小数点型に変換する
}
/// ## 11-1.エラー型の基本
/// ### リスト11.1 文字列を指定された型に変換する
/// ### 引数 value:変換対象文字列
/// ### 引数 conv: 変換種別
#[allow(dead_code)]
fn parse_01(value: String , conv: ValueConversion) {
    if conv == ValueConversion::Int {
        println!("{:?}" , value.parse::<i32>());
    }else{
        println!("{:?}" , value.parse::<f32>());
    }
}
/// ## 11-1.エラー型の基本
/// ### リスト11.1 文字列を指定された型に変換する
/// ### 関数の利用
#[allow(dead_code)]
pub fn use_parse_01() {
    parse_01(String::from("123"), ValueConversion::Int);
    parse_01(String::from("123"), ValueConversion::Float);
    parse_01(String::from("ABC"), ValueConversion::Int);
    parse_01(String::from("ABC"), ValueConversion::Float);
}
```

ンプルコードでは参考になりにくいので、スレッド間通信を利用した駅名検索機能を実装します。

## 12-4-1 サンプルコードの概要

サンプルコードでは2つのスレッドを起動し、チャネルを利用してデータ交換します（図12.1）。

図12.1 サンプルコードの仕様




Providerは予め駅名のJSONをデシリアライズし、Clientから駅名が送信されてくるのを待っています。Clientではキー入力された駅名をチャネルでProviderに送信し、Providerからの検索結果受信待ちにします。Providerは駅名を受信すると検索メソッドを使って駅名検索し、その結果をClientにチャネルを使って送信します。検索結果を受信したClientは結果を画面出力します。Clientは“end”が入力されるとProviderにそのまま送信して処理を終了します。Providerも”end”を受信すると処理を終了します。標準機能の場合もcrossbeamの場合も以下のような実行結果になります。JSONファイルをアクセスする機能は、サンプルプロジェクトに入っているので参考にしてください。

```
駅名を入力してください
新宿
"9番目の駅です"
駅名を入力してください
神田
"23番目の駅です"
駅名を入力してください
end
Client 終了
Provider 終了
```

## 12-4-2 Clientの実装（標準ライブラリ）

標準ライブラリでは、std::sync::mpscモジュールからデータ転送機能が提供されています。通常のスレッド間データ転送機能だけでなく、非同期処理を利用した転送機能も用意されています。

std::sync::mpsc::Sender<T>はデータ送信機能を、std::sync::mpsc::Receiver<T>が受信機能を提供し、共にジェネリック型で指定された型の値を扱うことができます。

channel()関数を利用すると生成されたSenderとReceiverをタプル形式で受け取ることができます。Senderのsend()メソッドで値を送信でき、Receiverのrecv()メソッドで受信ができます（**リスト12.10**）。

リスト12.10 標準ライブラリを利用したClientの実装（messaging_std.rs）

```
use std::sync::mpsc::{Sender, Receiver};
use chapter12::station::{Station,enter_station};
use chapter12::stations::Stations;
#[allow(dead_code)]
pub struct Client;
impl Client {
    /// ## 12-4 スレッド間通信
    /// ### リスト12.10 Providerに検索要求を出し、結果を受け取るメソッド
    /// ### 引数 Providerへの送信チャネル p_sender: Sender<String>
    /// ### 引数 Clinetの受信チャネル c_receiver: Receiver<String>
    #[allow(dead_code)]
    pub fn search_request(p_sender: Sender<String>,c_receiver: Receiver<String>) {
        loop {
            let entry_name = enter_station("駅名を入力してください");
            // 入力された駅名をProviderに送信
            p_sender.send(entry_name.clone())
                .unwrap_or_else(|error| println!("{:?}", error));
            // endなら終了
            if entry_name == "end" {
                break;
            }
            // Providerからの検索結果受信
            c_receiver.recv().and_then(|result| {
                    println!("{:?}", result);
                    Ok(())
                }).unwrap_or_else(|error| println!("{:?}", error));
        }
        println!("Client 終了");
    }
```

```
17:     }
```

```
値:10
値:10
値:20
値:20
値:30
値:30
値:40
値:40
値:50
値:50
合計1:150
合計2:150
```

# 12-9 タスク間通信

非同期タスクはスレッドに割り当てられて実行されるので、当然タスク間での通信が必要になるケースも多々あります。async_stdクレートもチャネルを利用した通信が可能です。

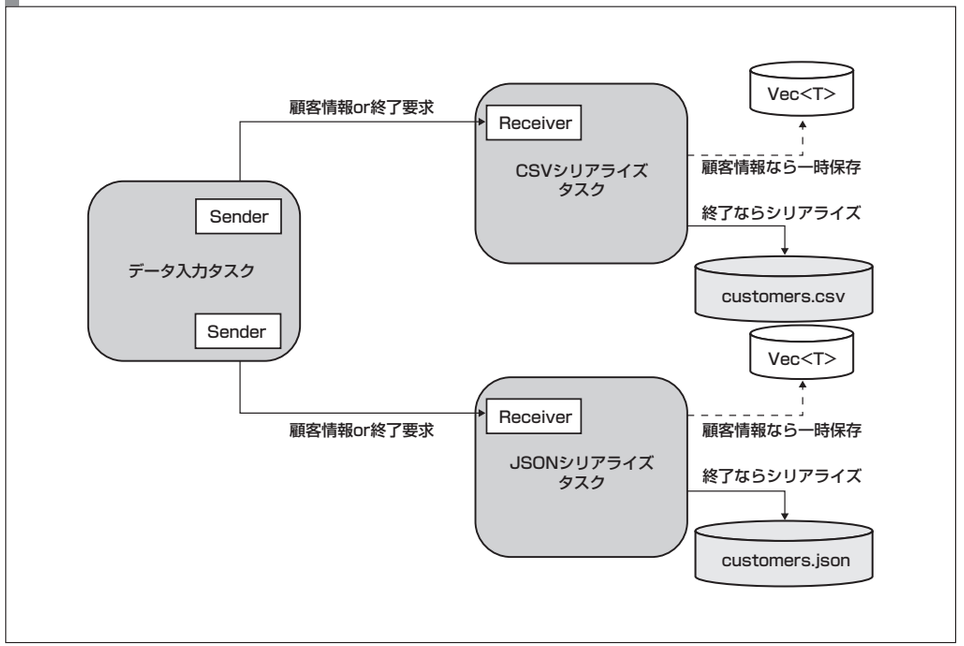
## 12-9-1 サンプルコードの概要

スレッドでのサンプルコードと同様にキー入力を受け付けるClient役は、CSV、JSON形式に受信データをシリアライズするタスクとチャネルで通信します。

通信する値は顧客の氏名とメールアドレスを持ったCustomer構造体のインスタンスです。シリアライズするタスクは顧客情報を受信すると常にベクタに値を格納することを繰り返します。氏名が“end”ならば送信終了とみなし、ファイルにシリアライズしてすべてのタスクは終了します。

このサンプルコードは、anyhowクレートのResult<T>型を利用して結果を返し、CSVファイルのアクセスにはcsvクレートを利用しているので、Cargo.tomlに「csv = “1.16”」の依存定義が必要です（**図12.4**）。

図12.4 タスク間通信サンプルコードの構成




## 12-9-2 Customer構造体

最初に通信する値から紹介していきます。顧客情報を扱う構造体を**リスト12.29**のように実装しています。

リスト12.29 Customer構造体の利用（customer.rs）

```
1:  use std::fmt::{Display, Formatter};
2:  use serde::Serialize;
3:  /// ## 12-9 タスク間通信
4:  /// ### リスト12.29 顧客情報構造体
5:  #[derive(Debug , Clone , Serialize)]
6:  pub struct Customer {
7:      name:   String, // 氏名
8:      email:  String  // メールアドレス
9:  }
10: impl Customer {
11:     pub fn new(_name:String,_email:String) -> Self{
12:         Self{name:_name , email:_email}
13:     }
```

# 13-4 ドキュメントテスト

Rustにはドキュメントテストというユニークなテスト方法があります。関数などのテストには大変便利なテスト方法です。

ドキュメントを記述する記号には、「//」、「/*～*/」、「///」、「//!」とありますが、「///」と「//!」はドキュメントテストを記述できます。この節では、ドキュメントテストの記述ルールと実装について解説します。

## 13-4-1 ドキュメントテストの構文

ドキュメントテストを記述する際に重要になるのが開始の記述です。コメント中に「```」を記述することで以降にドキュメントテストを記述できます。一般的には開始と終わりを「```」で囲みます。

```
/// ``` ドキュメントテストの開始
/// テストコード
/// テストコード
/// ```
```

## 13-4-2 ドキュメントテストの記述

ドキュメントテストの記述方法は、assert!()マクロなどを利用して結果を評価すればよいことは変わりません（**リスト13.13**）。

リスト13.13 ドキュメントテスト（document_test.rs）

```
1:  use crate::target::Guest;
2:  /// ## 13-4.ドキュメントテスト
3:  /// ### リスト13.13 ドキュメントテストの記述
4:  /// ### 年齢が10歳でキャンペーンでなければ500円を返すことを検証する
5:  /// ```
6:  /// use chapter13::document_test::calc_fee_case_01;
7:  /// let result = calc_fee_case_01();
8:  /// assert_eq!(500 , result);
9:  /// ```
10: pub fn calc_fee_case_01() -> u32 {
11:     let  guest = Guest::new(10 , false); // インスタンスを生成する
12:     let result = guest.clone().calc_fee().unwrap(); // 入園料を取得する
13:     result
14: }
```

別モジュールを作成し、calc_fee_case_01()という関数を実装しました。関数はGuest構造体のインスタンスを生成してcalc_fee()関数の実行結果を返しています。この関数をテストするのが6行目から始まるテストコードです。コメント内でcalc_fee_case_01()関数を実行した結果をassert_eq!()で評価しています。

ドキュメントテストを記述する場合、このブロックは別モジュールとみなされるため、useキーワードを利用してテスト対象を参照可能にします。ドキュメントテストも「cargo test」で実行でき、VS Codeの場合はコメントの上に[Run Doctest]が表示されるので、それをクリックしてテストを実行できます。

```
running 1 test
test src¥document_test.rs - document_test::calc_fee_case_01 (line 10) ... ok
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.32s
```

# 13-5 外部クレートの利用

テストをサポートする外部クレートも多数提供されています。標準テスト機能と組み合わせることで、テストドライバ実装を効率化し、分かりやすく実装できます。テストをサポートする外部クレートとして、simple_test_caseを解説します。

## 13-5-1 外部クレートの定義

テスト機能をサポートするクレートはCargo.tomlの[dev-dependencies]セクションに記述しなければなりません。[dev-dependencies]セクションは開発時に利用する外部クレートの依存定義をするセクションです（**リスト13.14**）。simple_test_caseクレートはバージョン1.1.0を利用します。

リスト13.14 依存定義（Cargo.toml）

```
1:  [package]
2:  name = "chapter13"
3:  version = "0.1.0"
4:  edition = "2021"
5:  …
6:  [dev-dependencies]
7:  simple_test_case = "1.1.0"
8:  [dependencies]
9:  …
```

# 16章 O/R Mapper

**これまでは、DBMS別のクレートをテーマに提供される機能や使い方について紹介してきました。次に、実際のアプリケーション開発で欠かすことのできないORMフレームワークについて見ていくことにしましょう。Rustでも2017から2018年頃にPostgresSQLやMySQLなどオープンなRDBMSを対象としたORMが提供されるようになりました。ORMにはDeiselなどさまざまなものが提供されています。本章ではSea ORMというO/R Mapperフレームワークを紹介します。**

## 16-1 O/R Mapperの概要

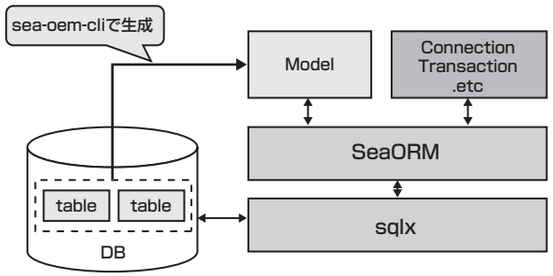
Sea ORMは多くのDBMSをサポートし、非同期処理をサポートしている大変多機能なORMです。この節ではSeaORMの概要、サンプルアプリケーションで利用しているクレート、アプリケーション作成の前処理となる事柄について解説します。

### 16-1-1 SeaORMの概要

SeaORMは最初から非同期処理をサポートしたORMとして開発されたもので、tokioやasync_std、actixなどの非同期ランタイムを選択して利用できます（**図16.1**）。本書では取り上げていませんがSQL実行基盤には、PoastgreSQL、MySQL、SQLite、SQLServerなど多数のRDBMSをサポートするSQL実行クレートのsqlxを利用しています。使用するデータベース環境を構築する機能や必要な構造体や列挙型を生成するためのCLI（Command Line Interface）が利用できます。

本節ではあらかじめ用意されたデータベースの利用を想定した解説をしていきます。

**図16.1 Sea ORMの概要**




SeaORM多様なデータベースアクセス機能を提供し、ドキュメントやサンプルコードも豊富です(注1)。機能が多く1つの章では紹介しきれないため、本章では基本的な使い方を解説します。

### 16-1-2 使用する外部クレート

本節で利用するサンプルコードではSeaORM以外に以下の外部クレートを利用します（**リスト16.1**）。

**リスト16.1 サンプルコードの依存定義**

```
1:  [dependencies]
2:  sea-orm = {version="0.10.6" , features=["sqlx-postgres" , "runtime-tokio-rustls" ,
    "macros"] , default-features = false}
3:  tokio = {version="1.17.0" , features=["full"]}
4:  serde = {version = "1.0.136", features = ["derive"] }
5:  serde_json = { version = "1.0.79", default-features = false, features = ["alloc"] }
6:  proc-macro2 = "1.0.37"
7:  lombok = "0.3.3"
8:  async-trait = "0.1.53"
9:  anyhow = "1.0.56"
10: dotenv = "0.15.0"
11: chrono = "0.4.19"
12: log = "0.4.16"
13: env_logger = "0.10.0"
```

2行目でSeaORMの依存定義をしています。Features属性で利用するデータベースドライバと非同期ランタイムを指定します。データベースドライバはsqlx-postgres、sqlx-mysql、sqlx-

(注1) https://docs.rs/sea-orm/0.10.6/sea_orm/