Appendix 2

Rust用語集

Rustを学んでいると、たくさんの聞き慣れない言葉を耳にします。ここでは、Rustに関連する用語をまとめてみます。本書を読んでいて分からない部分があったら、この用語集を参考にしてください。

Cargo

パッケージ管理システムであり、Rustプログラムのビルドツールでもあります。プログラムが依存しているライブラリーのダウンロードを行い、ライブラリーのビルドを行います。詳しくは2章1節で解説しています。

FizzBuzz問題

英語圏での言葉遊びであるFizz Buzzをプログラミング言語で記述する問題です。1章で詳しく解説します。

null / null安全

C言語やJavaなど多くのプログラミング言語で値がないことを表現するのにnullを使います。しかし、多くのシステムにnullを原因とする脆弱性が存在しています。nullを許容しない言語を「null安全」と呼び、Rustもnull安全な言語です。

TOML

設定ファイルを記述するための言語です。可読性が高いのが特徴で、必要最小限の機能を備えており、Cargoの設定ファイルはTOMLで記述することになっています。TOMLの仕様は、次のURLで公開されています。詳しくは、本書の2章01で解説しています。

TOMLの仕様書（日本語訳）
URL https://github.com/toml-lang/toml.io/blob/main/specs/ja/v1.0.0-rc.2.md

rustup

Rustプロジェクトが公式に提供しているインストーラーです。WindowsでRustを動かすためには、rustupのほか、Microsoftが提供するビルドツールをインストールする必要があります。

Unicode（ユニコード）

世界各国で使われている文字を一つの文字コードで表現するために、世界中の文字を収録する文字コード規格です。世界中のさまざまな言語の文字に対して通し番号を割り当てるため、Unicodeで記された文書には、日本語・中国語・韓国語・ロシア語など多くの言語の文字を混在させることができます。

UTF-8

Unicode/UCSで使える8ビット符号単位の文字符号化形式です。Rustの文字列では、UTF-8が標準の文字エンコーディングとして採用しています。ASCII文字と互換性を持たせるために、半角英数字を含むASCII部分は1バイト、その他の部分を2-6バイトで表現します。詳しくは、本書3章5節で解説しています。

Visual Studio Code / VSCode

原稿執筆時点で最もメジャーなRustの開発です。Microsoftが開発しているWindows、Linux、macOS用のソースコードエディターです。詳しくは1章で紹介しています。

Visual Studio CodeのWebサイト
URL https://code.visualstudio.com/

WSL / Windows Subsystem for Linux

Windows上でLinuxを実行するOS標準の機能です。WSL上でRustを動かすことも可能です。

イテレーター（iterator）

配列など複数要素を持つ集合的データ構造に対して、各要素の繰り返し処理を実現するために用いる抽象表現です。「反復子」と呼ぶこともあります。詳しくは4章で解説します。

インタープリター（interpreter）とコンパイラー（compiler）

コンピューターでプログラムを実行する方式の違いです。ソースコードを逐次解釈して実行する方式がインタープリターです。これに対して、プログラムを一度マシン語にコンパイルしてから実行する方式をコンパイラーと呼びます。Pythonはインタープリター言語で、Rustはコンパイラー言語です。なお、より詳しい内容を1章で解説しています。

オープンソース（open source）

ソースコードが公開されており、自由に利用できるソフトウェアのことです。

正確なオープンソースの定義は「ソースコードを商用、非商用の目的を問わず利用、修正、頒布することを許し、それを利用する個人や団体の努力や利益を遮ることがないことが求められてる」ものとなっています。

Rustもオープンソースのプロジェクトであり、ライセンスは「Apacheライセンスバージョン2.0」と「MITライセンス」のデュアルライセンスとなっています。

Rustのライセンスについて
URL `https://www.rust-lang.org/ja/policies/licenses`

型推論（type inference）

Rustのもつ強力な機能の1つで、コンパイラーが自動的にプログラムを解析して、変数や関数の型を適用する機能です。明示的に変数の宣言が不要になるので、コードの記述量を減らせる利点があり

ます。

ガベージコレクション（garbage collection）/ GC

プログラムが動的に確保したメモリ領域のうち、不要になった領域を自動的に解放する機能です。本書でRustと比較するプログラミング言語のPythonや、Java/C#/Goなど多くのプログラミング言語がガベージコレクションを採用しています。これに対して、Rustはガベージコレクションではなく、所有権システムを採用しています。

木構造（tree structure）

データ構造の1つです。代表的な利用例はOSのファイルシステムで、この木構造で表現されています。また、XMLやJSONなどのデータも木構造です。親となる要素が複数の子要素を持っており、子要素がさらに複数の孫要素を持つ構造です。木の枝が分かれている様子に似ていることから名付けられました。

クレート（crate）

Rustのモジュールは木構造で表現されており、モジュール群をまとめてクレートと呼びます。クレートはRustのライブラリーの基本単位です。詳しくは4章6節で解説しています。

クロージャー（英語:Closure）

敢えて関数を定義することなく関数を定義して利用するプログラミング言語の機能です。1章で解説します。

構造体（structure）

Rustにおけるデータ型の一つ。構造体を使うと複数の値をまとめて格納できます。構造体を使うことで、独自のデータ型を定義できます。詳しくは、3章4節で解説します。また、構造体を操作するメソッドが定義できます。メソッドについては4章1節で解説します。

参照（reference）

値そのものではなく、値を表す参照情報を参照として与えることができる機能です。Rustで変数の参照と言えば、変数に対する参照情報を意味します。C言語で言うところのポインターです。詳しくは1章と3章2節で解説しています。

参照外し（dereference）

参照から実際の値を得ることです。Rustでは参照を得るために「&変数」と記述し、参照外しを行うには「*変数」と記述します。詳しくは本書3章2節で紹介します。

借用（Borrow / Borrowing）

Rustの所有権システムの中で、値の所有権を一時的に貸し出す仕組みのことです。主に関数の呼び出しで行われます。関数の引数に参照を取ると、所有権が移動せず値を借用できます。

ジェネリクス（generics）

抽象的な型を指定してさまざまなデータ型の操作を可能にするプログラミング手法です。詳しくは4章で解説します。

所有権システム（ownership）

Rustのメモリ管理は所有権システムで行われます。メモリ管理の手法には、ガベージコレクションがあり、これはプログラムの実行中にメモリの不要な領域を検出し開放します。これに対して所有権システムでは、コンパイル時にメモリの確保と解放を管理します。そのためガベージコレクションよりも高速で効率よいのが特徴です。詳しくは、本書の3章で解説します。

スタック（stack）

スタックとは基本的なデータ構造の1つです。データを追加する（PUSH）と取り出す（POP）に対応したデータ構造です。データを追加（PUSH）する時は記憶領域の末尾にデータを追加して、取り出す（POP）時は記憶領域の末尾から値を取り出します。例えるなら、狭い机の上に資料を積んでいく状況に似ています。資料を A,B,C と積んでいった場 合、資料を取り出す時には、C,B,A と取っていかないと資料が崩れてしまいます。追加（PUSH）が資料を机の上に置くこと、取り出す（POP）が一番上にある資料を取り出すことに相当します。本書の4章ではスタックを利用してRPN電卓のプログラムを作成します。

スタック領域（stack memory）、ヒープ領域（heap memory）

スタック領域もヒープ領域も、実行時にプログラムが使用できるメモリの一部ですが、メモリの管理方法が異なります。「ヒープ領域」は自由にメモリの確保と解放が可能な領域です。これに足して「スタック領域」は、メモリを確保した順番でしかメモリを解放できないという制限を持つ領域です。Rustの所有権を学ぶには、この違いを覚えておく必要があります。詳しくは3章、および、6章で解説します。

スライス（slice）

Rustで配列や文字列・ベクター型のデータの一部を参照する型のことです。参照なので所有権がないのが特徴です。なお、文字列の参照を意味する&str型もスライスです。

スレッドセーフ（Thread-safe）

並列処理を行うプログラムを作成する際に、問題が生じない仕様や設計になっていることを言います。並列処理ではメモリの読み書きが同時に行われる可能性がありますが、これが原因でメモリ違反が発生することがあります。Rustでは並列処理を行う際、メモリ違反が起きないよう配慮されたライブラリーが用意されています。詳しくは本書の5章4節をご覧ください。

タプル（tuple）

タプルとは複数のデータを組にしてまとめて管理できるデータ型です。配列型と違って、異なる型のデータを組にできます。詳しくは、3章3節で解説します。

統合開発環境（IDE）

アプリケーション開発に必要なソフトウェアをまとめて1つにして提供するパッケージのことです。Rustの統合開発環境には、EclipseやIntelliJ IDEAが用意されています。とは言え、Rustの開発では必ずしも統合開発環境が必要となるわけではありません。詳しくは1章で紹介しています。

トレイト（trait）

共通の振る舞いを定義するためのもの。Rustにおけるトレイトは異なる型に対して共通の振る舞い（メソッド）を定義するのに使います。JavaやC#にある、インターフェイスの機能に似ています。詳しくは4章で解説します。

配列型（array）とベクター型（vector）

配列型もベクター型も複数の値をまとめて管理するのに利用します。Rustの配列型はサイズが固定であり後からサイズを変更することはできません。これに対して、ベクター型は、要素の追加削除が自由です。詳しくは、2章3節で解説します。

ハッシュマップ（hash map）/ 連想配列（associative array）

RustのHashMapを利用すると連想配列が実現できます。Pythonの辞書型のように、文字列から任意の値を取り出すことのできるデータ型です。2章05節で解説します。

標準入力（standard input）と標準出力（standard output）

標準入力とはプログラムが標準的に利用するデータ入力元で、同じように標準出力とはプログラムが標準的に利用するデータ出力先です。Rustをコマンドライン環境で使う場合、端末のコンソール画面（コマンドライン環境）からの入出力がこれに相当します。
Rustで標準出力に出力するには、println!を使います。また、標準入力から入力を得るには、std::io::stdin()で標準入力を得た後、read_lineメソッドを使います。詳しくは、2章4節をご覧ください。

ビット（bit）とバイト（byte）

1ビットとは、コンピューターが扱うデータの最小単位のことです。1ビットで表現できるのは0と1の2個の値だけです。そして1バイトは8ビットであり、8ビットあれば2の8乗個の値（0から255）を表現できます。詳しくは1章で解説しています。

並列処理（parallel processing）

プログラム内で複数の処理を同時に実行することを「並列処理」と呼びます。並列処理を行うと複数の処理を同時に実行させることができるため、コンピューターの処理速度を向上させるための手法としても有効です。特に、マルチコアプロセッサを搭載したシステムでは大幅な処理速度の向上が見込まれます。また、ネットワークのプログラムを作成する際には、並列処理が必須となります。詳しくは本書の5章4節をご覧ください。

マクロ（macro）

プログラムのソースコードをあらかじめ定義した規則に従って変換する機能です。ただし、Rustのマクロは単純なプログラムの置換処理に留まりません。マクロの詳細については6章で解説します。

ミュータブル（mutable）/イミュータブル（immutable）

ミュータブルは変更可能な変数であり、イミュータブルは変更可能な変数です。Rustでは「let 変数」を用いて変数を定義すると基本的にイミュータブルな変数となり、ミュータブルな変数を定義するには「let mut 変数」のようにミュータブルであることを明示する必要があります。

文字列リテラル（string literal）

プログラム内に記述される文字列を示す定数のことです。Rustではダブルクォートで、"abc…"のように記述したものが文字列リテラルとなります。定数なので不変であり変更できません。文字列や文字列リテラルについて詳しくは3章で解説します。

ライフタイム（lifetime）

Rustでは値の参照が有効な範囲(スコープ)を表します。ライフタイム を表現するには、<'a> のようなライフタイム注釈記法を記述します。詳しくは3章5節で紹介しています。

ラムダ式（lambda expressions）

敢えて関数を定義することなく関数を定義して利用するプログラミング言語の機能です。無名関数とも呼ばれており、Rustではクロージャーと呼びます。