

エラーメッセージと解決のヒント

Rustの学習を始めると、多くの方がエラーに悩まされます。安全で効率のよい実行結果を得るためには仕方がないのですが、やはりエラーの多さに戸惑う人も多いことでしょう。

ここでは、そうしたエラーのうち、よく陥りやすいパターンについて、事例と対処方法をピックアップしてみました。

Rust なぜ動かない？

Rustは安全性を開発目標のひとつに掲げています。そのため、コンパイル時にプログラムの問題を指摘する機構が優れています。コンパイルさえ無事に通れば、安全にプログラムを動かすことができます。

ただし、これは裏を返せば、Rustの所有権システムやその挙動を理解していないと、エラーが頻発し、コンパイルさえままならない、ひいてはプログラムを動かすこともできないことを意味しています。

ここでは、エラーメッセージと共に動きそうで動かないRustのプログラムを紹介し、どのように直したら良いのかを解説します。

●文字列リテラルでエラーが出る

【事例】

以下のように文字列を画面に表示するプログラムを作りました。しかし、エラーが出て動きません。どうしたら良いでしょうか。

参照するファイル `file: src/apx1/str_quote_err.rs`

```
// 問題のあるプログラム
fn main() {
    // 以下でエラーが出る
    let s = 'Hello, World!';
    println!("{}", s);
}
```

コンパイルすると以下のような「unterminated character literal(文字リテラルが閉じられていない)」というエラーが出ます。

```
~/repos/book-rust/data/src/apx1 % rustc str_quote_err.rs
error[E0762]: unterminated character literal
  --> str_quote_err.rs:4:27
4 |     let s = 'Hello, World!';
  |                                     ^^
error: aborting due to previous error
```

文字リテラルのエラーが出る

【対処方法】

本書で、Rustでは文字と文字列を区別する必要がある点を学びましょう。

「文字」は'a'のようにシングルクォートで表現し、「文字列」は"abc"のようにダブルクォートで表現します。以下のようにシングルクォートをダブルクォートに修正すれば、問題なくコンパイルが通ります。本書のChapter1で詳しく解説しています。

参照するファイル file: src/apx1/str_quote_fix.rs

```
fn main() {
    let s = "Hello, World!";
    println!("{}", s);
}
```

● i64型にi32型の値を代入できない

【事例】

i32型(32ビット整数)の値をi64型(64ビット整数型)に代入しようとしてしました。しかし、エラーが出て代入できません。i64の方がi32よりも大きいので、C言語などでは問題なく代入できるのですが、どうしてエラーになるのでしょうか。

参照するファイル file: src/apx1/i64_i32_err.rs

```
fn main() {
    // i32型の変数を定義
    let n: i32 = 100;
    // i64型にi32型の値を代入
    let m: i64 = n; // ←ここでエラー
    println!("{}", n, m);
}
```

コンパイルすると以下のエラーが出ます。

```
fish /Users/kijirames/Book-rust/data/src/apx1
~/b/d/s/apx1 >>> rustc int_to_str_err.rs
error[E0106]: missing lifetime specifier
  --> int_to_str_err.rs:3:30
   |
3  | fn int_to_str(value: i64) -> &str {
   |                             ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value with an elided lifetime, but the lifetime cannot be derived from the arguments
   help: consider using the 'static' lifetime
3  | fn int_to_str(value: i64) -> &'static str {
   |                             ^^^^^^^^^
error: aborting due to previous error

For more information about this error, try `rustc --explain E0106`.
~/b/d/s/apx1 >>>
```

型エラーが出る

【対処方法】

Rustは型に厳しい言語です。同じ整数型であっても暗黙的な型変換は行われません。下記のようにasを利用して型変換を明示することで変換できます。データ型については本書のChapter1で解説しています。

参照するファイル file: src/apx1/i64_i32_fix.rs

```
fn main() {
    // i32型の変数を定義
    let n: i32 = 100;
    // i64型にi32型の値を代入
    let m: i64 = n as i64; // asでi64に変換
    println!("{}", n, m);
}
```

●文字列の参照を戻り値として返したい

整数を文字列に変換し、その参照を返す次のようなint_to_str関数を定義してみました。しかしコンパイルできません。どのように直せば良いのでしょうか。

参照するファイル file: src/apx1/int_to_str_err.rs

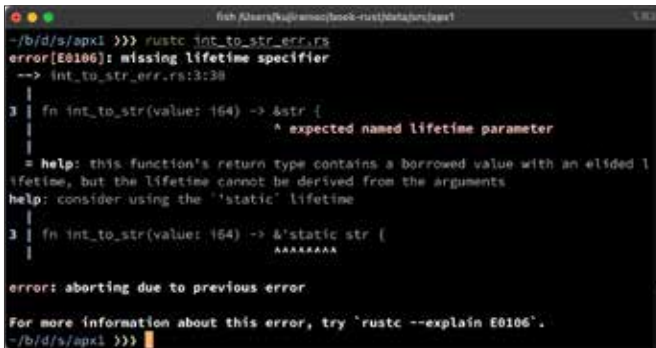
```
// 問題のあるプログラム
fn int_to_str(value: i64) -> &str {
    let s = format!("{}", value);
```

```

    return &s;
}
fn main() {
    let s = int_to_str(256);
    println!("{}", s);
}

```

このコードをコンパイルすると次のようなエラーが表示されます。赤字のエラーを確認すると「expected named lifetime parameter(名前付きのライフタイムパラメーターが必要)」と表示されています。



```

fish /Users/kaji/miniconf3/book-rust/data/apx1 1.8.2
~/b/d/s/apx1 >>> rustc int_to_str_err.rs
error[E0106]: missing lifetime specifier
  --> int_to_str_err.rs:3:38
   |
3  | fn int_to_str(value: i64) -> &str {
   |                               ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value with an elided lifetime, but the lifetime cannot be derived from the arguments
   help: consider using the 'static' lifetime
   |
3  | fn int_to_str(value: i64) -> &'static str {
   |                               ^^^^^^^^^
   |
error: aborting due to previous error

For more information about this error, try `rustc --explain E0106`.
~/b/d/s/apx1 >>>

```

ライフタイムに関するエラーが出る

【対処方法】

まずは、Rustの所有権システムに対する理解を深めましょう。表示されているエラーメッセージを確認すると、関数が参照を返す場合にはライフタイム注釈記法の指定が必要となることを表しています。

しかし、このプログラムの問題は、Rustの所有権システムにあります。この関数ではString型の変数sを作成し、その参照を関数の戻り値として返そうとしています。ところが、変数sは関数の終了とともに破棄されてしまうため、sの参照は無効となってしまいます。この点に関して、Chapter3の所有権と参照についての部分で詳しく解説しています。

この関数を正しく動かすには、所有権が正しく移動できるように、戻り値の型を&strではなく、Stringを返すようにします。

参照するファイル file: src/apx1/int_to_str_fix.rs

```

// 整数を文字列に変換する関数
fn int_to_str(value: i64) -> String {
    let s = format!("{}", value);
}

```

```

    s
}
fn main() {
    let s = int_to_str(256);
    println!("{}", s);
}

```

コンパイルして実行してみましょう。参照型は所有権が移動しませんが、String型であれば所有権が移動するので、String型を戻り値とすることで正しく値を返すことができます。詳しくは書籍のChapter3で解説しています。

```

$ rustc int_to_str_fix.rs && ./int_to_str_fix
256

```

● 文字列を分割して文字列のベクターを作りたい

【事例】

カンマで区切られた文字列をVec<String>型に変換しようと思って以下のプログラムを作りましたが動きません。どうしたら良いでしょうか。

参照するファイル file: src/apx1/split_err.rs

```

// 問題のあるプログラム
fn main() {
    let target = "aaa,bbb,ccc";
    let lines:Vec<String> = target.split(",").collect();
    println!("{}", lines);
}

```

このプログラムをコンパイルしようとするすると下記のようなエラーが出ます。

```

~/h/p/2/b/c/s/apx1 >>> rustc split_err.rs
error[E0277]: a value of type 'Vec<String>' cannot be built from an iterator over elements of type '&str'
  -> split_err.rs:3:47
   |
3  |         let lines:Vec<String> = target.split(",").collect();
   |                                     ^^^^^^^^^ value of type 'Vec<String>' cannot be built from 'std::iter::Iterator<Item=&str>'
   = help: the trait 'FromIterator<&str>' is not implemented for 'Vec<String>'

error: aborting due to previous error

For more information about this error, try 'rustc --explain E0277'.
~/h/p/2/b/c/s/apx1 >>>

```

splitの後のcollectがおかしいと指摘が出る

よくよく見ると、エラー部分について、親切にヘルプが表示されています。「the trait `FromIterator<&str>` is not implemented for `Vec<String>`」（トレイト `FromIterator<&str>` は `Vec<String>` を実装していません）。

つまり、String::splitメソッドはFromIterator<&str>を返すのですが、Vec<String>型に変換する機能はないということです。

Pythonなどのスクリプト言語であれば、当然splitメソッドは文字列のリストを返すのですが、Rustのsplitメソッドは、文字列を分割するためのイテレーターを返します。加えて、分割した文字列はStringではなく参照型の&strを返すのです。

そこで、for文やmapメソッドを利用して次のように修正できます。

参照するファイル file: src/apx1/split_fix.rs

```

fn main() {
    let target = "aaa,bbb,ccc";
    // for文を使ってVec<String>に変換
    let mut lines = vec![];
    for line in target.split(",") {
        lines.push(line.to_string());
    }
    println!("{:?}", lines);
    // mapを使ってVec<String>に変換
    let lines:Vec<String> =
        target.split(",").map(|s| s.to_string()).collect();
    println!("{:?}", lines);
}

```

実行すると、splitメソッドを使って文字列を分割し、Vec<String>型のデータを得られます。

```
$ rustc split_fix.rs && ./split_fix
["aaa", "bbb", "ccc"]
["aaa", "bbb", "ccc"]
```

● 可変参照を連続で使いたい

【事例】

次のプログラムのように、ミュータブルな文字列sの参照を2回連続で使いたいのですが、2回目の参照でエラーになってしまいます。どのように修正したら良いのでしょうか。

参照するファイル file: src/apx1/ref2_err.rs

```
// 問題のあるプログラム
fn main() {
    // 文字列を生成
    let mut s = String::from("能ある鷹は爪を隠す");
    // 参照を得たい
    let ref1 = &mut s;
    let ref2 = &mut s; // ←ここでエラーになる
    // 画面に表示
    println!("ref1={}, ref2={}", ref1, ref2);
}
```

このプログラムをコンパイルしようすると下記のようなエラーが出ます。

```
~/repos/book-rust/data/src/apx1 % rustc ref2_err.rs
error[E0499]: cannot borrow 's' as mutable more than once at a time
--> ref2_err.rs:7:16
6 |     let ref1 = &mut s;
  |               ----- first mutable borrow occurs here
7 |     let ref2 = &mut s;
  |               ^^^^^^ second mutable borrow occurs here
8 |     // 画面に表示
9 |     println!("ref1={}, ref2={}", ref1, ref2);
  |                                     ---- first borrow later used here
error: aborting due to previous error
```

2つ目の参照でエラーが出る

【対処方法】

Rustではメモリの競合を防ぐため同時に2つ以上の可変参照を利用したアクセスが禁じられています。

す。そこで、可変参照を利用したいスコープを分割することで問題を解決できます。以下のように修正すると良いでしょう。

参照するファイル file: src/apx1/ref2_fix.rs

```
fn main() {  
    // 文字列を生成  
    let mut s = String::from("能ある鷹は爪を隠す");  
    // 連続で可変参照を使うにはスコープを分割する  
    {  
        let ref1 = &mut s;  
        println!("ref1={}", ref1);  
    } // ここでref1は破棄される  
    {  
        let ref2 = &mut s;  
        println!("ref2={}", ref2);  
    }  
}
```

実行すると正しく可変参照を得られて、参照内容を表示できます。

```
$ rustc ref2_fix.rs && ./ref2_fix  
ref1=能ある鷹は爪を隠す  
ref2=能ある鷹は爪を隠す
```