

Rust の String と &str の違いを chatGPT を使って理解してみた

表 1: Caption {#tbl:label}

a	b	c
1	2	3
4	5	6

追記: String の参照 &String が &str と同様に扱える場合について、今後記事を作成する予定。

はじめに

Rust の String 構造体と文字列リテラル &str の違いが可変長かどうかくらいしか分かっていなかったの、chatGPT に聞きつつ理解しようと思った。人生初のプログラミング言語として学部一年の時に C 言語をやったので、データ構造がイメージしやすかったと思う。

結論

- String の実態は Vec<u8>
- &str の実態は &[u8]
- &[u8] と &str の決定的な違いは u8 が表す数値が何のエンコードかを表す情報を持つかどうか

as_ref() と & 演算子

as_ref() という関数を見かけたので、これがなんなのかを ChatGPT に聞くところから始まった。予想としては + 演算子を数値以外の型で使えるようにするには Add トレイトの add() を定義すれば可能なので、それと同じように & 演算子が使用された時の振る舞いを定義できるのかと考えた。

■自分

変数.as_ref と &変数は同義？

■ChatGPT

変数.as_ref() と **&変数** は同じ結果を返すことがありますが、厳密には同じ意味ではありません。

- **&変数** は、単純に変数への不変の参照を作成します。これは元の変数と同じデータを指す新しい参照を作成します。変数の所有権は移動しません。これは値に対する不変の参照を取得するために使用されます。
- **変数.as_ref()** は、as_ref メソッドが実装されている型に対してのみ使用できます。このメソッドは元の値を新しい参照型に変換しようと試みるものです。これは、元の値を新しい型への参照として扱うことができる場合に有用です。例えば、文字列型 String を &str に変換する場合に使います。as_ref は元の値を変更せず、新しい参照を作成するだけです。

したがって、**変数.as_ref()** は、元の型が as_ref メソッドを実装しており、かつそのメソッドが新しい型への変換を提供する場合にのみ使用できます。一方、**&変数** はその型が不変の参照をサポートする限り、ほとんどの型で使用できます。

どうやら違うようだ。よく考えてみたら、as_ref() で & の振る舞いを定義したとしたら、ライブラリで as_ref() を見かけるようなことも起きないはず。なぜなら & で同じことができるのにわざわざ文字数を多く書く as_ref() を使う理由がないからだ。

■自分

&は変数のアドレス、as_ref は独自に定義した参照で、その実装として&が使われた場合は通常の&と同義である。であって？

■ChatGPT

実際の挙動を説明します。& は値への不変の参照を作成します。as_ref は、特定の型に対して元の値を新しい型への参照に変換しようとする方法を提供するトレイトです。通常、このトレイトは&self メソッドを介して & による参照を返します。この場合、**&変数** と **変数.as_ref()** は通常同じ意味になります。

例えば、標準ライブラリの String 型では、as_ref は&str への変換に使われます。そのため、**変数.as_ref()** は **&変数** と同じく String を &str に変換します。

しかし、as_ref は他の型に対しても使われます。具体的な挙動は型やトレイトの実装に依存します。要するに、as_ref は型の変換を行う方法を提供し、その方法は型ごとに異なることがあります。一般的には、as_ref は & による参照を返すことが多いですが、必ずしも同じ意味ではないことに注意してください。

大体予想通りだったが、ここで一番気になったのが**変数.as_ref()**は**&変数**と同じくStringを**&str**に変換します。という言葉だ。つまりas_ref()を使わなくてもString構造体の参照**&変数**が**&str**と同じであると言っているということだ。

String と &str のデータ構造

String

String 構造体は Vec<u8> と長さを基本構造としていると思っていた。簡単に書くと、

```
struct String {  
    vec: Vec<u8>,  
    len: usize,  
}
```

という感じ。だから & 演算子で参照するとそのまま vec<u8> と len をもった構造体自体の参照が得られると思っていた。しかし ChatGPT は&String は&str と同義だと言っている。C 言語ではポインタという 1 種類で表すが、Rust では参照とか生ポインタとか種類があるので、C 言語の考えをそのまま持ち込むとおかしくなりそうだが、この辺は C++ を知っていればもっと理解できたのかもしれない。

ChatGPT はすぐに嘘をつくので Rust の実装 を見てみた。as_ref() で検索をかけてそれっぽものを見つけられた。

```
impl AsRef<str> for String {  
    #[inline]  
    fn as_ref(&self) -> &str {  
        self  
    }  
}
```

```
impl AsRef<[u8]> for String {  
    #[inline]  
    fn as_ref(&self) -> &[u8] {  
        self.as_bytes()  
    }  
}
```

&self って自分自身の参照を返してるだけじゃん。意味わからん。

&str

&str は次のような構造体だと予想していた。

```
struct str {
    vec: &[u8],
    len: usize,
}
```

impl str にするか impl &str にするか迷うあたり自分が Rust を全く理解できていないと感じた。

■自分

String は実態として文字列を vec で保持するが、&str は通常の配列と同様に連続したデータ構造？

■ChatGPT

はい、正しいです。String は UTF-8 エンコードされた文字列データをバイトのベクタ (Vec<u8>) として保持します。このため、String は可変で、可変サイズの文字列データを格納できます。

一方、&str は不変の文字列スライスで、通常は連続したバイトのデータ構造です。&str は元のデータを参照し、そのデータは通常 &[u8] として表現されます。このスライスは文字列の一部分を指し示すため、文字列のサイズを示す情報を含むことが多いです。また、UTF-8 エンコードの文字列に対しても正しく作用します。

この違いは String が可変で所有権を持つ一方、&str は不変で借用を表すことから生じます。String は文字列データを生成、変更、所有するための型であり、&str はその一部または全部を参照するための型です。

こちらも大体予想通りだった。もう一つ気になる事があったので聞いてみた。

■自分

スライスと配列の違いは？

■ChatGPT

スライス (Slice) と配列 (Array) は、Rust において異なるデータ構造です。以下にそれぞれの違いを説明します：

1. 配列 (Array)：

- 固定サイズ：配列は宣言時に固定サイズを指定する必要があり、そのサイズは変更できません。
- データはスタックに格納：配列の要素は通常、スタックメモリに連続して格納されます。
- 所有権：配列はコピー可能なデータ型であり、その値を他の変数に代入すると、元の配列のコピーが作成されます。

2. スライス (Slice) :

- 可変サイズ：スライスは可変サイズのデータを参照するためのデータ構造で、配列やベクタなどのコレクションから部分的なデータを指し示します。
- データはヒープに格納：スライスは通常、ヒープメモリに格納されたデータの一部を参照します。
- 所有権を持たず、借用：スライスは元のデータの所有権を持たず、不変または可変な借用を提供します。これにより、元のデータを変更せずにその一部にアクセスできます。

スライスは、配列やベクタの特定の部分にアクセスするために非常に便利で、イテレーションやデータの部分的な処理に使用されます。また、スライスは配列を参照することで効率的なメモリ管理を提供します。

The Book より次の例を見つけた。

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

この例を見ると、`&str` は参照している先頭アドレスと長さを保持している事がわかる。先程簡易的な自分と思う `&str` の構造を書いたが、迷ってもおかしくはないと思えてきた。

String から `&str` への変換

大体のデータ構造が明らかになったので、String から `&str` への変換を考えてみる。

■自分

`&str` は `u8` 配列への参照だが、`string` は `vec` である。そして `vec` はヒープ上の配列への参照である。つまり `string` から `&str` を作り出すときラッピングを 2 回外しているということ？

■ChatGPT

はい、正確な説明です。String は `Vec<u8>` で、その内部データはヒープ上に格納されたバイトの配列です。`&str` は `u8` のバイトスライスへの不変な参照で、元の String データへのアクセスを提供します。したがって、String から `&str` を作成する際、ラッピングを 2 回外していると言えます。

具体的には、String は `Vec<u8>` へのラッパーであり、`&str` は `u8` のバイトスライスへのラッパーです。`&str` は元の String データを直接指すのではなく、バイトスライスとして参照します。このようなラッピングを介して、文字列データへの効率的なアクセスが実現されます。

このようなラッピングを外すような作業が 2 回もあるにも関わらず、`&String` が `&str` と同義であるらしい。

また、Vec は可変長というが、実際にはヒープ領域ののある程度のサイズをもった配列への参照をもっており、その配列がいっぱいになると新たにより大きな領域をヒープに確保して、そこにデータをコピーしていくという仕組みになっている。つまり String が持つ Vec がいっぱいになると新たにメモリを確保してそこにデータを移してしまうから、その String から作られた &str の参照先は存在しなくなってしまうのではないかという疑問が生まれた。

■自分

vec も内部では u8 の配列を持っていて string の変更があった場合、仮に長さを超えた場合は新たにヒープに配列を作り出す？

■ChatGPT

はい、String 内部では Vec<u8> が文字列データを格納しており、このデータはヒープ上に配置されます。String は不定長の文字列を格納でき、文字列が変更されると必要に応じてヒープ上に新しい配列が確保されます。

例えば、String に新しい文字を追加した場合、既存のヒープ上のデータに新しい文字を追加することはできません。その代わりに、新しい長さのデータを格納するために新しいヒープ領域にメモリが確保され、元のデータはコピーされることなく、新しいデータが追加されます。このプロセスは透過的に行われ、ユーザーには気にする必要はありません。

この方法により、String は可変な文字列を効率的に扱えます。ヒープ上でのメモリ再割り当てやデータのコピーが必要な場合でも、Rust の所有権システムにより、データの整合性と安全性が保たれます。

■自分

その場合変更が行われた string から作り出した&str はアドレスが変わっているから使えくない？

■ChatGPT

String から作成した &str は、元の String が変更されても利用可能です。なぜなら、&str は元のデータへの不変な参照であり、元のデータが変更されても&str 自体は有効で、元のデータを参照し続けます。

変更が行われても &str が無効になるのは、その参照の寿命（ライフタイム）が元のデータよりも短い場合です。通常、&str の寿命は元の String と同じかそれよりも長くなります。したがって、元の String が変更されても、適切にスコープが管理されていれば、&str を引き続き使用できます。

しかし、&str を作成した後で元の String が破棄される場合、&str は無効になります。そのため、&str を使用する際には、元のデータとの寿命に注意を払う必要があります。

Rust の所有権システムによって、変数に参照されなくなった値は自動的に破棄されるが、`&str` 変数によって参照されていれば、もともと `String` が持っていた `Vec` が持っていた配列の領域は破棄されないらしい。これは ChatGPT が違うことを言っている可能性が高いので、最後にテストしてみる。

`&[u8]` と `&str` の違い

`Vec` の実態はヒープ上に確保された `[u8]` だが、っそれを同じように参照している `&[u8]` と `&str` の違いは何か。

■自分

`&str` は `u8` 配列への参照と長さを持った別名とも言えるね

■ChatGPT

はい、その表現は正確です。`&str` は `u8` バイトの配列への不変な参照と、その配列の長さ情報を持つデータ型です。`&str` は文字列データを不変な形式で扱うためのデータ構造であり、UTF-8 エンコードされた文字列のバイト列を表現します。したがって、`&str` を `u8` バイトの配列への参照と見なすこともできます。

やはりどちらも `[u8]` 配列を参照していることには変わりないようだ。

■自分

`&str` と `&[u8]` の違いは実装されているメソッドくらいしか差がない？

■ChatGPT

`&str` と `&[u8]` はいくつかの面で類似していますが、重要な違いも存在します。以下は主な違いと類似点です:

違い:

1. キャラクターエンコーディング: 重要な違いの1つは、`&str` が UTF-8 エンコードの文字列を表現するのに対し、`&[u8]` はバイトのスライスであり、文字列のエンコーディング情報を持たないことです。したがって、`&str` は文字列としての操作（例: 文字数、文字列比較など）を行う際にエンコーディングに対処できますが、`&[u8]` はバイトの配列としての操作に限定されます。
2. メソッド: `&str` は文字列操作のための多くのメソッド（`len()`、`chars()`、`contains()` など）を提供しますが、`&[u8]` はバイトデータとしての操作に適したメソッド（`len()`、`iter()` など）を提供します。そのため、用途に合わせてメソッドを選択することができます。

類似点:

1. スライス: どちらも元のデータをスライスして参照し、元のデータに対する変更を行わずにデータにアクセスできます。両方ともスライスのような操作をサポートしており、データの一部を表現するために使用できます。
2. 長さ情報: 両方の型はスライスの長さ情報を保持しており、範囲外アクセスを防ぐのに役立ちます。
3. 不変性: 両方の型は不変であり、元のデータを変更しません。したがって、参照元のデータは不変であることを保ちつつ、スライスを通じてデータにアクセスできます。

どちらの型を使用するかは、特定の用途やデータの種類の依存し、文字列データを扱う場合は通常 `&str` を使用し、バイトデータを扱う場合は `&[u8]` を使用することが一般的です。

■自分

決定的な違いはその 8 ビットの値が utf8 なのか shiftjis なのかとかのエンコードの情報の変数を持つてことだね

■ChatGPT

正確です。`&str` と `&[u8]` の決定的な違いは、エンコーディングに関する情報です。`&str` は UTF-8 エンコードの文字列データを表現し、エンコーディングに関する情報を持っています。一方、`&[u8]` はバイトのスライスであり、エンコーディング情報を持たず、文字列のエンコーディングについては知識がありません。

この違いにより、`&str` は文字列としての操作やエンコーディングに関連したタスクに適していますが、`&[u8]` はバイトデータとしての操作に適しています。どちらを使用するかは、操作対象のデータの性質に依存します。

メソッドなどの違いもちろんあるが、決定的な違いはエンコードの情報を持つかどうかだということが分かった。

実際に試してみた

`String` と `&str` の違いが分かったので、実際に試してみた。

1. `&str` が参照する `String` が破棄された後でも `&str` が参照する先のデータが残っているかどうか

```
let s: &str;

{
    let string = String::from("hello");
```



```
s = &string[..];
}
```

```
println!("{}", s)
```

これはやはりコンパイルエラーになった。

2. String が持つ Vec が持つ [u8] の場所がサイズオーバーで変更された場合

次の方法で Vec の最大容量を取り出せる。

```
let inner_vec = Vec::<u8>::with_capacity(2);
```

これを String 構造体の vec に代入することで String が持つ Vec の最大容量を変更できるが、それが可能なメソッドは用意されていないらしい。

String 構造体に次のようなメソッドを発見した。

```
pub fn capacity(&self) -> usize {
    self.vec.capacity()
}
```

これで String 構造体の vec が持つ Vec の最大容量を取り出せる。

```
let s1 = String::from("ab");
let s2 = String::from("abcd");
```

```
println!("{}", s1.capacity());
println!("{}", s2.capacity());
```

```
2
```

```
4
```

String::from() で使った文字列リテラルの大きさになった。String の vec の大きさはピッタリサイズなのだろうか。

次に既存の String に文字列を追加した

```
let mut s = String::from("ab");
s.push_str("cd");
```

```
println!("{}", s.capacity());
```

```
8
```

s の vec の最大容量が s の文字列の長さを超えたことが確認できた。つまりこの s.push_str() する前に s から作られた slice が有効かどうかでこの問題を検証できる。

```
let mut s = String::from("ab");
let str: &str = &s[..];
s.push_str("cd");

println!("{}", str);
println!("{}", s.capacity());
```

結果はコンパイルエラー。

ここで重大な見落としが発覚した。Rust ではそもそも、その変数が参照されているうちはその変数が持つ値を変更できない。つまり今まで疑問に思ってきたことは、Rust では起こりえないことだったのだ。

まとめ

String と &str の違いをある程度理解する事ができた。また、Rust という言語が極めて安全で、自分が疑問に思っているようなことは言語仕様ですでに淘汰されていることがわかった。

&String が &str になる理由はまだ理解できてないので、もう少し調べてみたい。