CとRust

目的

- 組込みCエンジニア向け
  Rustの宣伝

内容

Cの典型的な苦手分野：パーサー

Rustの場合

ベンチマーク

ライフタイム

まとめ

Cの典型的な苦手分野：パーサー

```c
char **split(const char *text, const char delimiter)
{
    const char *s, *p;
    char **vstr;
    int i;

    vstr = malloc(sizeof(*vstr) * strlen(text));
    i = 0;

    for (s = p = text; *p; p++) {
        if (*p == delimiter) {
            vstr[i++] = strndup(s, p - s);
            s = p + 1;
        }
    }

    vstr[i++] = strndup(s, p - s);
    vstr[i] = NULL;

    return vstr;
}
```

おそい

"de:ad:be:ef:01:23"

↓

split()

↓

"de",
"ad",
"be",
"ef",
"01",
"23"

Cの典型的な苦手分野：パーサー

```c
char **split(const char *text, const char delimiter)
{
    const char *s, *p;
    char **vstr;
    int i;

    vstr = malloc(sizeof(*vstr) * strlen(text));
    i = 0;

    for (s = p = text; *p; p++) {
        if (*p == delimiter) {
            vstr[i++] = strndup(s, p - s);
            s = p + 1;
        }
    }

    vstr[i++] = strndup(s, p - s);
    vstr[i] = NULL;

    return vstr;
}
```

⬅ Q. こんなコード書かんでしょ？

A. よくあります

参考文献：g_strsplit (glib)

```
2393  gchar**
2394  g_strsplit (const gchar *string,
2395              const gchar *delimiter,
2396              gint         max_tokens)
2397  {
2398    GSList *string_list = NULL, *slist;
2399    gchar **str_array, *s;
2400    guint n = 0;
2401    const gchar *remainder;
2402
2403    g_return_val_if_fail (string != NULL, NULL);
2404    g_return_val_if_fail (delimiter != NULL, NULL);
2405    g_return_val_if_fail (delimiter[0] != '\0', NULL);
2406
2407    if (max_tokens < 1)
2408      max_tokens = G_MAXINT;
2409
2410    remainder = string;
2411    s = strstr (remainder, delimiter);
2412    if (s)
2413      {
2414        gsize delimiter_len = strlen (delimiter);
2415
2416        while (--max_tokens && s)
2417          {
2418            gsize len;
2419
2420            len = s - remainder;
2421            string_list = g_slist_prepend (string_list,
2422                                           g_strndup (remainder, len));
2423            n++;
2424            remainder = s + delimiter_len;
2425            s = strstr (remainder, delimiter);
2426          }
2427      }
```

https://github.com/GNOME/glib/blob/glib-2.25.7/glib/gstrfuncs.c#L2394
https://developer.gimp.org/api/2.0/glib/glib-String-Utility-Functions.html#g-strsplit

g_strsplit ()

```
gchar**             g_strsplit                          (const gchar *string,
                                                         const gchar *delimiter,
                                                         gint max_tokens);
```

Splits a string into a maximum of *max_tokens* pieces, using the given *delimiter*. If *max_tokens* is reached, the remainder of *string* is appended to the last token.

As a special case, the result of splitting the empty string "" is an empty vector, not a vector containing a single string. The reason for this special case is that being able to represent a empty vector is typically more useful than consistent handling of empty elements. If you do need to represent empty elements, you'll need to check for the empty string before calling `g_strsplit()`.

string : a string to split.

delimiter : a string which specifies the places at which to split the string. The delimiter is not included in any of the resulting strings, unless *max_tokens* is reached.

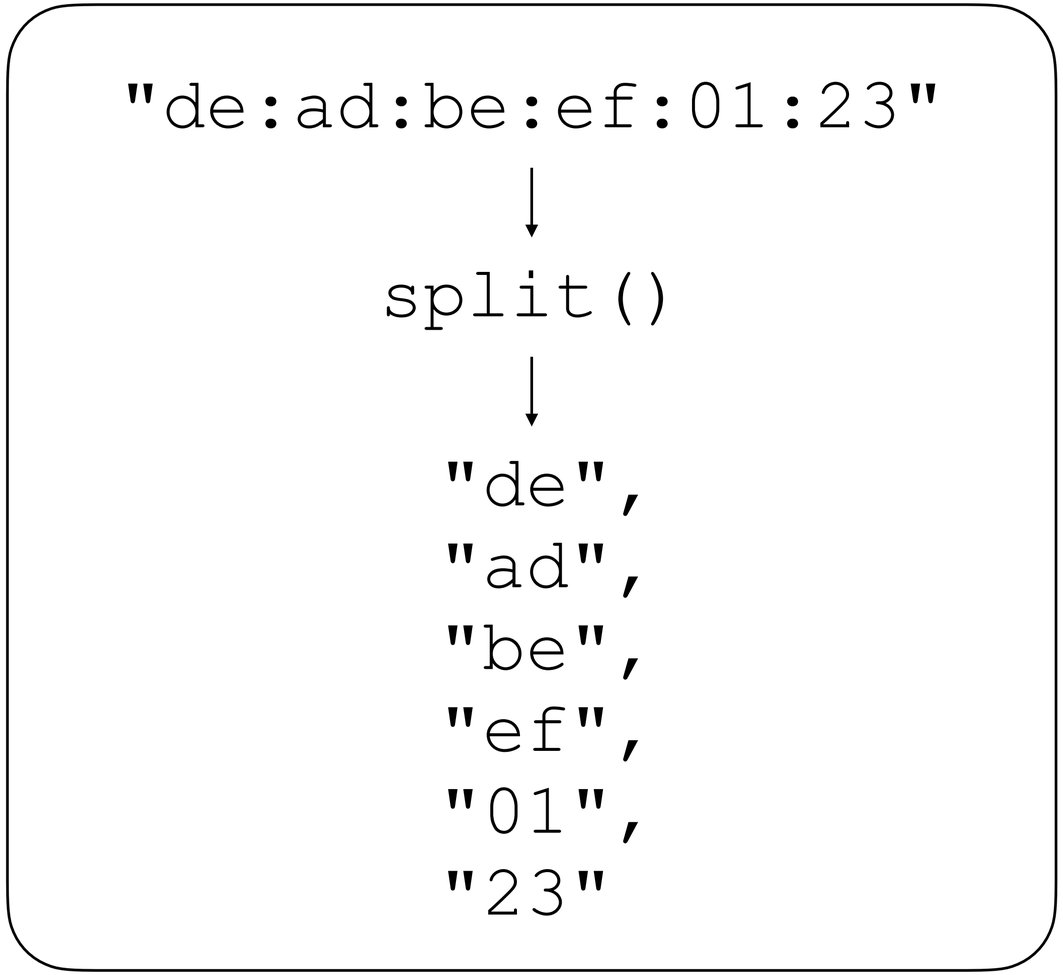
max_tokens : the maximum number of pieces to split *string* into. If this is less than 1, the string is split completely.

Returns : a newly-allocated `NULL`-terminated array of strings. Use `g_strfreev()` to free it.

Rustの場合

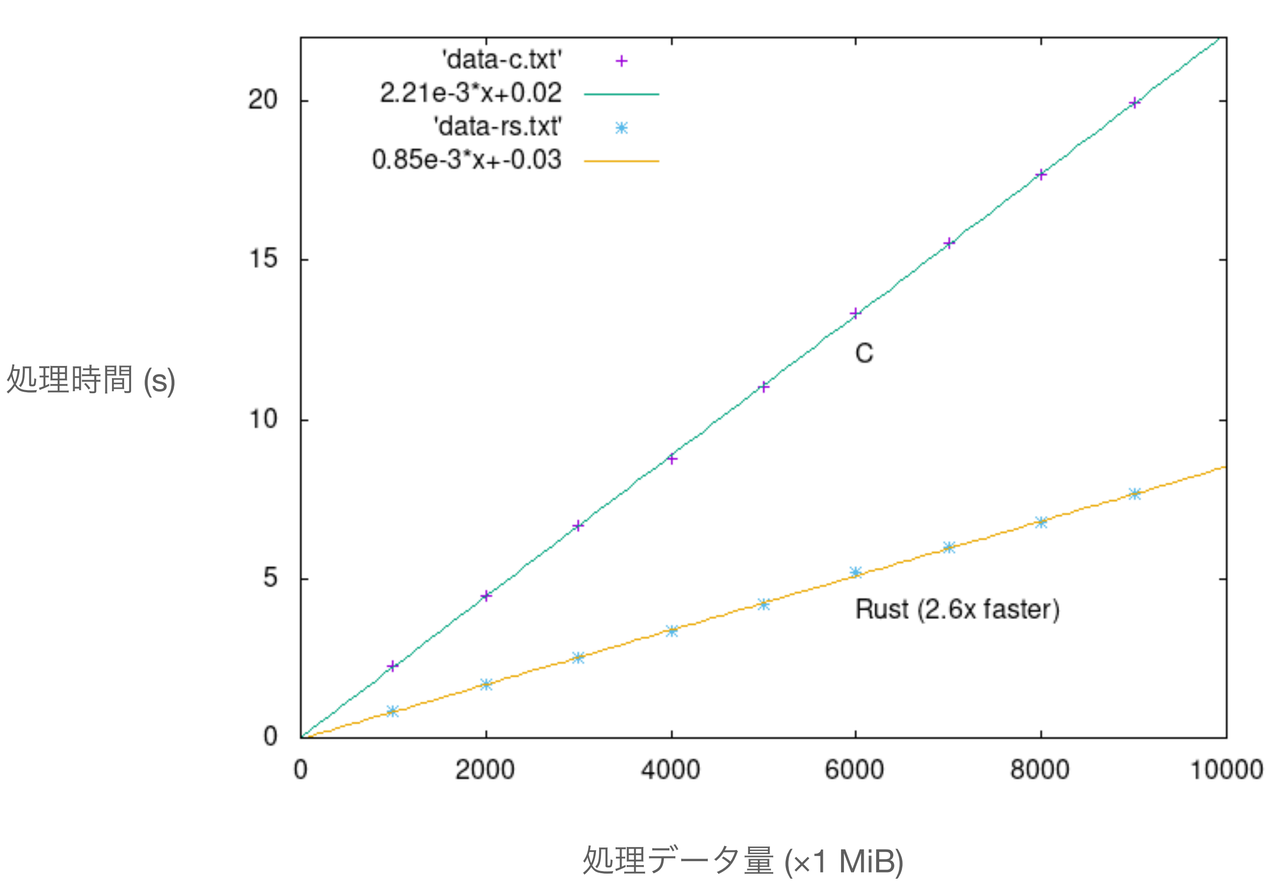
```rust
fn main() {
    let a = "de:ad:be:ef:01:23";
    for i in a.split(":") {
        println!("{}", i);
    }
}
```

はやい



ベンチマーク

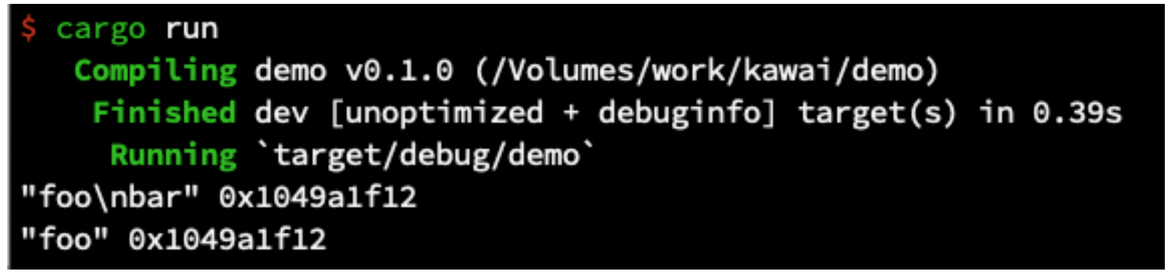
https://github.com/takumak/split-benchmark

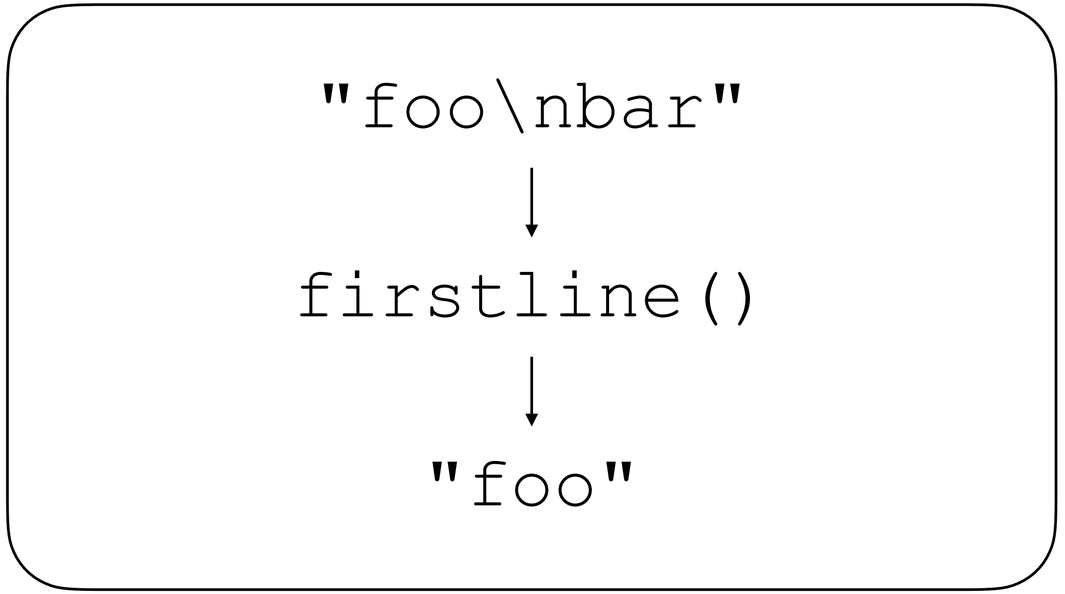


- データはChaCha8Rng同一シードで生成
- Cはg_strsplit相当の実装 (strndup)
- Rustはstd::str::Split
- Cは全部コピーしてるので当然遅い
- 条件がフェアじゃないが
  「この言語ではこう書くことが多い」
  を比べている
- これは極端な例で、Rustのほうが遅いことも多い

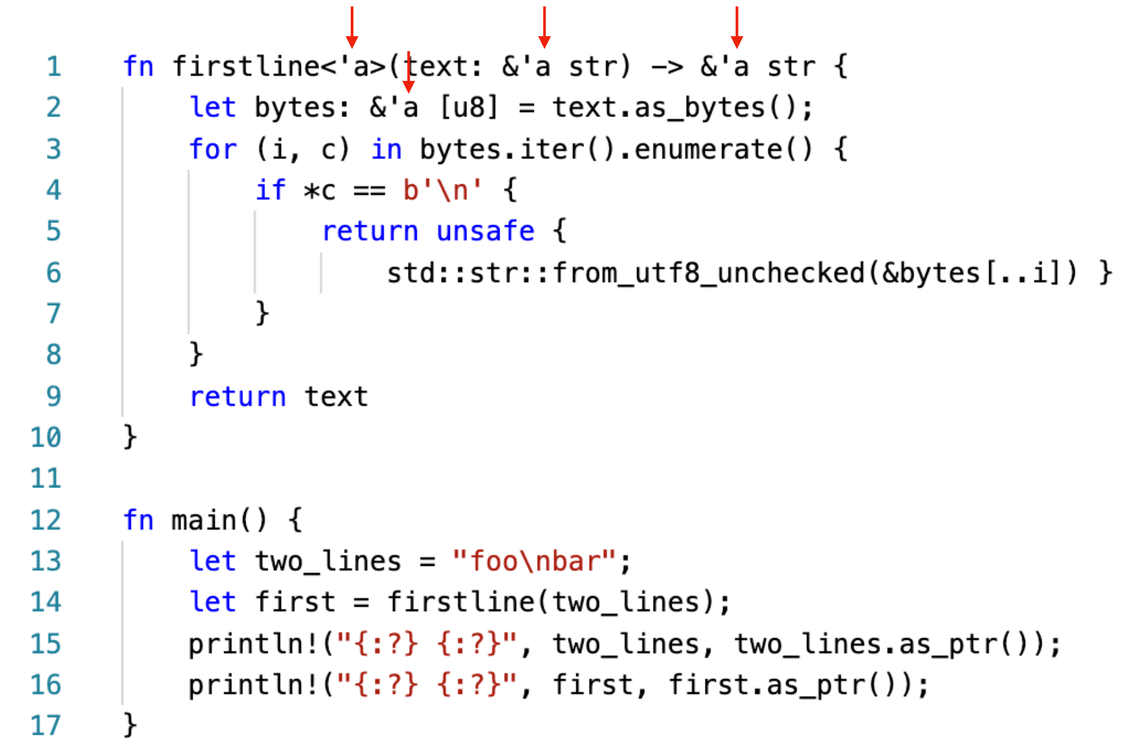
ライフタイム

```
1   fn firstline<'a>(text: &'a str) -> &'a str {
2       let bytes: &'a [u8] = text.as_bytes();
3       for (i, c) in bytes.iter().enumerate() {
4           if *c == b'\n' {
5               return unsafe {
6                   std::str::from_utf8_unchecked(&bytes[..i]) }
7           }
8       }
9       return text
10  }
11
12  fn main() {
13      let two_lines = "foo\nbar";
14      let first = firstline(two_lines);
15      println!("{:?} {:?}", two_lines, two_lines.as_ptr());
16      println!("{:?} {:?}", first, first.as_ptr());
17  }
```





ライフタイム

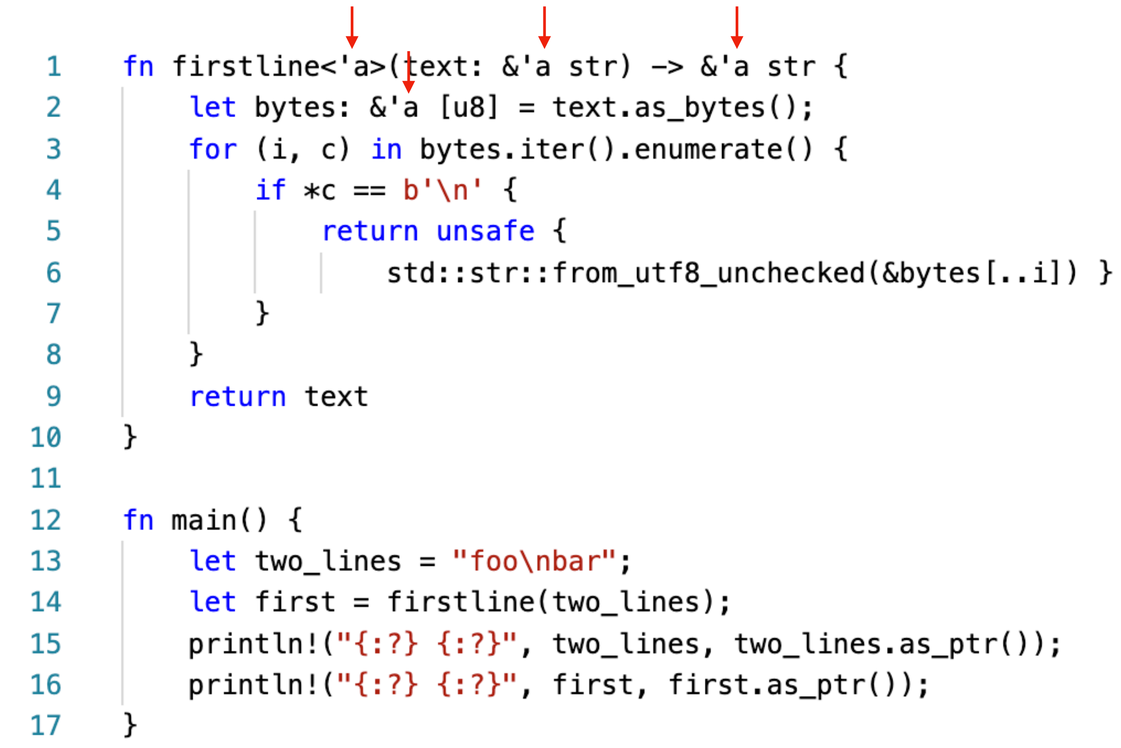


```
1   fn firstline<'a>(text: &'a str) -> &'a str {
2       let bytes: &'a [u8] = text.as_bytes();
3       for (i, c) in bytes.iter().enumerate() {
4           if *c == b'\n' {
5               return unsafe {
6                   std::str::from_utf8_unchecked(&bytes[..i]) }
7           }
8       }
9       return text
10  }
11
12  fn main() {
13      let two_lines = "foo\nbar";
14      let first = firstline(two_lines);
15      println!("{:?} {:?}", two_lines, two_lines.as_ptr());
16      println!("{:?} {:?}", first, first.as_ptr());
17  }
```

```
$ cargo run
   Compiling demo v0.1.0 (/Volumes/work/kawai/demo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
     Running `target/debug/demo`
"foo\nbar" 0x1049a1f12
"foo" 0x1049a1f12
```

- `'a`で第一引数`text`とローカル変数`bytes`と返り値の生存区間が同じであると宣言している
(ライフタイムという)
- `firstline`の呼び出し元(`main`)は引数と返り値が同じ区間で生存してないといけないとわかる
- `firstline`は生存区間を宣言することで、引数をコピーせず、参照を返すことができる
- Cでは`main`と`firstline`が互いを信頼しないので、`firstline`は`malloc`してコピーする実装が普通である

ライフタイム



```
1   fn firstline<'a>(text: &'a str) -> &'a str {
2       let bytes: &'a [u8] = text.as_bytes();
3       for (i, c) in bytes.iter().enumerate() {
4           if *c == b'\n' {
5               return unsafe {
6                   std::str::from_utf8_unchecked(&bytes[..i]) }
7           }
8       }
9       return text
10  }
11
12  fn main() {
13      let two_lines = "foo\nbar";
14      let first = firstline(two_lines);
15      println!("{:?} {:?}", two_lines, two_lines.as_ptr());
16      println!("{:?} {:?}", first, first.as_ptr());
17  }
```

```
$ cargo run
   Compiling demo v0.1.0 (/Volumes/work/kawai/demo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.39s
     Running `target/debug/demo`
"foo\nbar" 0x1049a1f12
"foo" 0x1049a1f12
```

- ライフタイムのチェックはビルド時に行う
- つまり参照カウンタのCoWより速い

まとめ

カプセル化と高速は

トレードオフじゃなかった

おわり