Chapitre 17. Chaînes et texte

> *La chaîne est une structure de données rigide et partout où elle est transmise, il y a beaucoup de duplication de processus. C'est un véhicule parfait pour cacher des informations.*
>
> —Alan Perlis, épigramme #34

Nous avons utilisé le texte principal de Rusttypes, `String`, `str`, et `char`, tout au long du livre. Dans "Types de chaînes", nous avons décrit la syntaxe des littéraux de caractères et de chaînes et montré comment les chaînes sont représentées en mémoire. Dans ce chapitre, nous couvrons plus en détail la gestion de texte.

Dans ce chapitre:

- Nous vous donnons quelques informations sur Unicode qui devraient vous aider à comprendre la conception de la bibliothèque standard.
- Nous décrivons le `char` type, représentant un seul point de code Unicode.
- Nous décrivons les types `String` et , représentant des séquences de caractères Unicode possédées et empruntées. `str` Ceux-ci ont une grande variété de méthodes pour construire, rechercher, modifier et itérer sur leur contenu.
- Nous couvrons les fonctionnalités de formatage de chaîne de Rust, comme les macros `println!` et `. format!` Vous pouvez écrire vos propres macros qui fonctionnent avec des chaînes de formatage et les étendre pour prendre en charge vos propres types.
- Nous donnons un aperçu du support des expressions régulières de Rust.
- Enfin, nous expliquons pourquoi la normalisation Unicode est importante et montrons comment le faire dans Rust.

Quelques arrière-plans Unicode

Ce livreconcerne Rust, pas Unicode, qui a déjà des livres entiers qui lui sont consacrés. Mais les types de caractères et de chaînes de Rust sont conçus autour d'Unicode. Voici quelques extraits d'Unicode qui aident à expliquer Rust.

ASCII, Latin-1 et Unicode

Unicodeet ASCIImatch pour tous les points de code ASCII, de `0` à `0x7f` : par exemple, les deux attribuent au caractère `*` le point de code `42` . De même, Unicode attribue aux mêmes caractères que le jeu `0` de `0xff` caractères ISO / CEI 8859-1, un sur-ensemble de huit bits d'ASCII à utiliser avec les langues d'Europe occidentale. Unicode appelle cette plage de points de code le *bloc de code Latin-1*, nous désignerons donc ISO/IEC 8859-1 par le nom plus évocateur *Latin-1* .

Comme Unicode est un sur-ensemble de Latin-1, la conversion de Latin-1 en Unicode ne nécessite même pas de table :

```
fn latin1_to_char(latin1: u8) ->char {
    latin1 as char
}
```

La conversion inverse est également triviale, en supposant que les points de code se situent dans la plage Latin-1 :

```
fn char_to_latin1(c: char) ->Option<u8> {
    if c as u32 <= 0xff {
        Some(c as u8)
    } else {
        None
    }
}
```

UTF-8

La rouille `String` et `str` les types représentent du texte en utilisant l'encodage UTF-8formulaire. UTF-8 encode un caractère sous la forme d'une séquence de un à quatre octets (Figure 17-1).

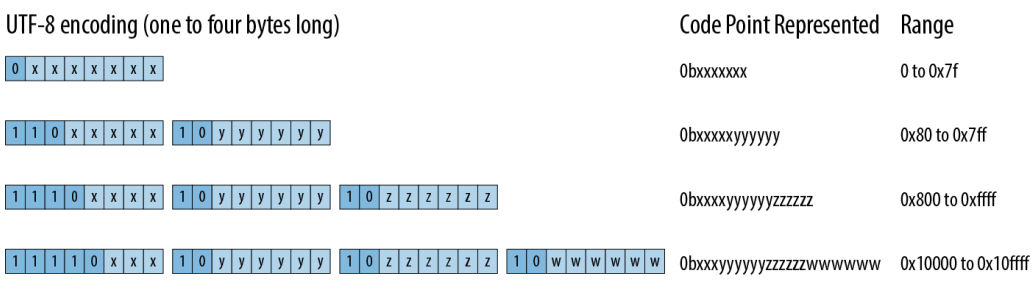

Illustration 17-1. L'encodage UTF-8

Il existe deux restrictions sur les séquences UTF-8 bien formées. Premièrement, seul le codage le plus court pour un point de code donné est considéré comme bien formé ; vous ne pouvez pas passer quatre octets à encoder un point de code qui tiendrait dans trois. Cette règle garantit qu'il

existe exactement un encodage UTF-8 pour un point de code donné. Deuxièmement, UTF-8 bien formé ne doit pas encoder les nombres de bout en `0xd800` bout `0xdfff` ou au-delà `0x10ffff` : ceux-ci sont soit réservés à des fins autres que les caractères, soit entièrement en dehors de la plage d'Unicode.

La figure 17-2 montre quelques exemples.

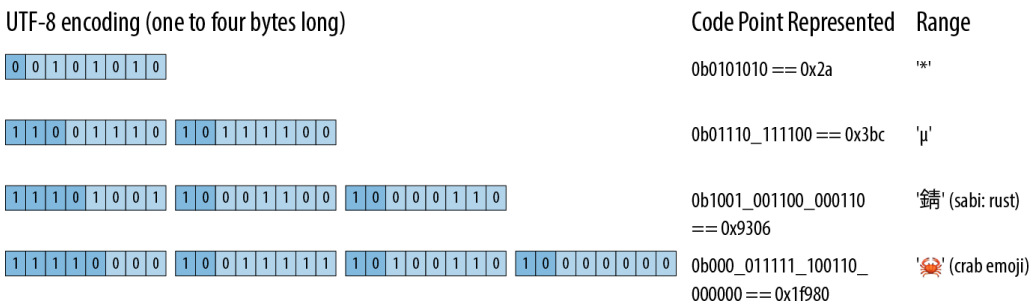

Illustration 17-2. Exemples UTF-8

Notez que, même si l'emoji crabe a un encodage dont l'octet de tête ne contribue que par des zéros au point de code, il a toujours besoin d'un encodage à quatre octets : les encodages UTF-8 à trois octets ne peuvent transmettre que des points de code de 16 bits, et `0x1f980` est de 17 peu de temps.

Voici un exemple rapide d'une chaîne contenant des caractères avec des encodages de longueurs variables :

```
assert_eq!("うどん: udon".as_bytes(),
           &[0xe3, 0x81, 0x86, // う
             0xe3, 0x81, 0xa9, // ど
             0xe3, 0x82, 0x93, // ん
             0x3a, 0x20, 0x75, 0x64, 0x6f, 0x6e // : udon
           ]);
```

La figure 17-2 montre également quelques propriétés très utiles d'UTF-8 :

- Étant donné que UTF-8 encode les points de code `0` à travers `0x7f` comme rien de plus que les octets `0` à `0x7f`, une plage d'octets contenant du texte ASCII est UTF-8 valide. Et si une chaîne UTF-8 ne comprend que des caractères ASCII, l'inverse est également vrai : l'encodage UTF-8 est un ASCII valide.
  Il n'en va pas de même pour Latin-1 : par exemple, Latin-1 encode `é` comme l'octet `0xe9`, ce que UTF-8 interpréterait comme le premier octet d'un encodage à trois octets.
- En regardant les bits supérieurs de n'importe quel octet, vous pouvez immédiatement dire s'il s'agit du début de l'encodage UTF-8 d'un ca-

ractère ou d'un octet au milieu d'un.
- Seul le premier octet d'un encodage vous indique la longueur totale de l'encodage, via ses bits de tête.
- Étant donné qu'aucun encodage ne dépasse quatre octets, le traitement UTF-8 ne nécessite jamais de boucles illimitées, ce qui est agréable lorsque vous travaillez avec des données non fiables.
- En UTF-8 bien formé, vous pouvez toujours dire sans ambiguïté où commence et se termine l'encodage des caractères, même si vous partez d'un point arbitraire au milieu des octets. Les premiers octets UTF-8 et les octets suivants sont toujours distincts, de sorte qu'un encodage ne peut pas commencer au milieu d'un autre. Le premier octet détermine la longueur totale de l'encodage, donc aucun encodage ne peut être le préfixe d'un autre. Cela a beaucoup de belles conséquences. Par exemple, la recherche d'un caractère délimiteur ASCII dans une chaîne UTF-8 ne nécessite qu'un simple balayage de l'octet du délimiteur. Il ne peut jamais apparaître comme une partie d'un encodage multi-octets, il n'est donc pas du tout nécessaire de suivre la structure UTF-8. De même, les algorithmes qui recherchent une chaîne d'octets dans une autre fonctionneront sans modification sur les chaînes UTF-8, même si certains n'examinent même pas chaque octet du texte recherché.

Bien que les encodages à largeur variable soient plus compliqués que les encodages à largeur fixe, ces caractéristiques rendent UTF-8 plus confortable à utiliser que vous ne le pensez. La bibliothèque standard gère la plupart des aspects pour vous.

Directionnalité du texte

Alors que les scriptscomme le latin, le cyrillique et le thaï sont écrits de gauche à droite, d'autres scripts comme l'hébreu et l'arabe sont écrits de droite à gauche. Unicode stocke les caractères dans l'ordre dans lequel ils seraient normalement écrits ou lus, de sorte que les octets initiaux d'une chaîne contenant, par exemple, du texte hébreu encodent le caractère qui serait écrit à droite:

```
assert_eq!("ערב טוב".chars().next(), Some('ע'));
```

Caractères (char)

Un Rust `char` est une valeur 32 bitstenant un point de code Unicode. A `char` est assuré de se situer dans la plage de `0` à `0xd7ff` ou dans la plage

`0xe000` de `0x10ffff` ; toutes les méthodes de création et de manipulation de `char` valeurs garantissent que cela est vrai. Le `char` type implémente `Copy` and `Clone` , ainsi que tous les traits habituels de comparaison, de hachage et de formatage.

Une tranche de chaîne peut produire un itérateur sur ses caractères avec `slice.chars()` :

```
assert_eq!("カ二".chars().next(), Some('カ'));
```

Dans les descriptions qui suivent, la variable `ch` est toujours de type `char` .

Classer les caractères

Le `char` type a des méthodes pour classercaractères dans quelques catégories courantes, comme indiqué dans le Tableau 17-1 . Ceux-ci tirent tous leurs définitions d'Unicode.

Tableau 17-1. Méthodes de classification pour le `char` type

Méthode	La description	Exemples
`ch.is_numeric()`	Un caractère numérique. Cela inclut les catégories générales Unicode « Nombre ; chiffre » et « Nombre ; lettre » mais pas « Chiffre ; autre".	`'4'.is_numeric()` `'十'.is_numeric()` `'⑧'.is_numeric()`
`ch.is_alphabetic()`	Un caractère alphabétique : la propriété dérivée "Alphabétique" d'Unicode.	`'q'.is_alphabetic()` `'七'.is_alphabetic()`
`ch.is_alphanumeric()`	Soit numérique, soit alphabétique, comme défini précédemment.	`'9'.is_alphanumeric()` `'餡'.is_alphanumeric()` `!'*'.is_alphanumeric()`
`ch.is_whitespace()`	Un caractère d'espacement : propriété de caractère Unicode "WSpace=Y".	`' '.is_whitespace()` `'\n'.is_whitespace()` `'\u{A0}'.is_whitespace()`

Méthode	La description	Exemples
`ch.is_control()`	Un caractère de contrôle : la catégorie générale "Autre, contrôle" d'Unicode.	`'\n'.is_control()` `'\u{85}'.is_control()`

Un ensemble parallèle de méthodes se limite à ASCIIuniquement, retournant `false` pour tout non-ASCII `char` (Tableau 17-2).

Tableau 17-2. Méthodes de classification ASCII pour `char`

Méthode	La description	Exemples
`ch.is_ascii()`	Un caractère ASCII : un dont le point de code se situe entre `0` et `127` inclus.	`'n'.is_ascii()` `!'ñ'.is_ascii()`
`ch.is_ascii_alphabetic()`	Une lettre ASCII majuscule ou minuscule, dans la plage `'A'..='Z'` ou `'a'..='z'`.	`'n'.is_ascii_alphabetic()` `!'1'.is_ascii_alphabetic()` `!'ñ'.is_ascii_alphabetic()`
`ch.is_ascii_digit()`	Un chiffre ASCII, dans la plage `'0'..='9'`.	`'8'.is_ascii_digit()` !'-'.is_ascii_digit() `!'⑧'.is_ascii_digit()`
`ch.is_ascii_hexdigit()`	N'importe quel caractère dans les plages `'0'..='9'`, `'A'..='F'` ou `'a'..='f'`.	
`ch.is_ascii_alphanumeric()`	Un chiffre ASCII ou une lettre majuscule ou minuscule.	`'q'.is_ascii_alphanumeric()` `'0'.is_ascii_alphanumeric()`

Méthode	La description	Exemples
`ch.is_ascii_control()`	Un caractère de contrôle ASCII, y compris 'DEL'.	`'\n'.is_ascii_control()` `'\x7f'.is_ascii_control()`
`ch.is_ascii_graphic()`	Tout caractère ASCII qui laisse de l'encre sur la page : ni un espace ni un caractère de contrôle.	`'Q'.is_ascii_graphic()` `'~'.is_ascii_graphic()` `!' '.is_ascii_graphic()`
`ch.is_ascii_uppercase()`, `ch.is_ascii_lowercase()`	Lettres majuscules et minuscules ASCII.	`'z'.is_ascii_lowercase()` `'Z'.is_ascii_uppercase()`
`ch.is_ascii_punctuation()`	Tout caractère graphique ASCII qui n'est ni alphabétique ni un chiffre.	
`ch.is_ascii_whitespace()`	Un caractère d'espacement ASCII : un espace, une tabulation horizontale, un saut de ligne, un saut de page ou un retour chariot.	`' '.is_ascii_whitespace()` `'\n'.is_ascii_whitespace()` `!'\u{A0}'.is_ascii_whitespace()`

Toutes les `is_ascii_` ... méthodessont également disponibles sur le `u8` type d'octet :

```
assert!(32u8.is_ascii_whitespace());
assert!(b'9'.is_ascii_digit());
```

Faites attention lorsque vous utilisez ces fonctions pour implémenter une spécification existante comme une norme de langage de programmation ou un format de fichier, car les classifications peuvent différer de manière surprenante. Par exemple, notez que `is_whitespace` et `is_ascii_whitespace` diffèrent dans leur traitement de certains caractères :

```
let line_tab = '\u{000b}'; // 'line tab', AKA 'vertical tab'
assert_eq!(line_tab.is_whitespace(), true);
assert_eq!(line_tab.is_ascii_whitespace(), false);
```

La `char::is_ascii_whitespace` fonction implémente une définition d'espace blanc commune à de nombreuses normes Web, alors que `char::is_whitespace` suit la norme Unicode.

Manipulation des chiffres

Pour manipuler les chiffres, vous pouvez utiliser les méthodes suivantes :

`ch.to_digit(radix)`

Décidesi `ch` est un chiffre en base `radix` . Si c'est le cas, il renvoie `Some(num)` , où `num` est un `u32` . Sinon, ça revient `None` . Ceci ne reconnaît que l'ASCIIchiffres, et non la classe plus large de caractères couverte par `char::is_numeric` . Le `radix` paramètre peut aller de 2 à 36. Pour les bases supérieures à 10, les lettres ASCII des deux cas sont considérées comme des chiffres avec des valeurs comprises entre 10 et 35.

`std::char::from_digit(num, radix)`

Une fonction gratuitequi convertit la `u32` valeur numérique `num` en a `char` si possible. Si `num` peut être représenté par un seul chiffre dans `radix` , `from_digit` renvoie `Some(ch)` , où `ch` est le chiffre. Lorsque `radix` est supérieur à 10, `ch` il peut s'agir d'une lettre minuscule. Sinon, ça revient `None` .

C'est l'inverse de `to_digit` . Si `std::char::from_digit(num, radix)` est `Some(ch)` , alors `ch.to_digit(radix)` est

`Some(num)`. Si `ch` est un chiffre ASCII ou une lettre minuscule, l'inverse est également vrai.

`ch.is_digit(radix)`

Retour `true` si `ch` est un chiffre ASCII en base `radix`. Cela équivaut à `ch.to_digit(radix) != None`.

Ainsi, par exemple :

```
assert_eq!('F'.to_digit(16), Some(15));
assert_eq!(std:: char:: from_digit(15, 16), Some('f'));
assert!(char::is_digit('f', 16));
```

Conversion de casse pour les caractères

Pour gérer la casse des caractères:

`ch.is_lowercase()`, `ch.is_uppercase()`

Indiquers'il `ch` s'agit d'un caractère alphabétique minuscule ou majuscule. Celles-ci suivent les propriétés dérivées des minuscules et des majuscules d'Unicode, elles couvrent donc les alphabets non latins comme le grec et le cyrillique et donnent également les résultats attendus pour l'ASCII.

`ch.to_lowercase()`, `ch.to_uppercase()`

Reveniritérateurs qui produisent les caractères des équivalents minuscules et majuscules de `ch`, selon les algorithmes de conversion de cas par défaut Unicode :

```
let mut upper = 's'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);
```

Ces méthodes renvoient un itérateur au lieu d'un seul caractère car la conversion de casse en Unicode n'est pas toujours un processus un à un :

```
// The uppercase form of the German letter "sharp S" is "SS":
let mut upper = 'ß'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);

// Unicode says to lowercase Turkish dotted capital 'İ' to 'i'
// followed by `'\u{307}'`, COMBINING DOT ABOVE, so that a
```

```
// subsequent conversion back to uppercase preserves the dot.
let ch = 'İ'; // `'\u{130}'`
let mut lower = ch.to_lowercase();
assert_eq!(lower.next(), Some('i'));
assert_eq!(lower.next(), Some('\u{307}'));
assert_eq!(lower.next(), None);
```

> Pour plus de commodité, ces itérateurs implémentent le `std::fmt::Display` trait, vous pouvez donc les transmettre directement à une macro `println!` ou `.write!`

Conversions vers et depuis des entiers

`as` Opérateur de Rustconvertira a `char` en n'importe quel entiertapez, masquant silencieusement tous les bits supérieurs :

```
assert_eq!('B' as u32, 66);
assert_eq!('饂' as u8, 66);   // upper bits truncated
assert_eq!('二' as i8, -116); // same
```

L' `as` opérateur convertira n'importe quelle `u8` valeur en a `char` , et `char` implémentera `From<u8>` également, mais des types entiers plus larges peuvent représenter des points de code invalides, donc pour ceux que vous devez utiliser `std::char::from_u32` , quirenvoie `Option<char>` :

```
assert_eq!(char:: from(66), 'B');
assert_eq!(std:: char:: from_u32(0x9942), Some('饂'));
assert_eq!(std:: char::from_u32(0xd800), None); // reserved for UTF-16
```

Chaîne et chaîne

Rouille `String` et `str` typessont garantis pour tenir seulement bien forméUTF-8. La bibliothèque garantit cela en limitant les façons dont vous pouvez créer des valeurs `String` et `str` les opérations que vous pouvez effectuer sur celles-ci, de sorte que les valeurs soient bien formées lors de leur introduction et le restent lorsque vous travaillez avec elles. Toutes leurs méthodes protègent cette garantie : aucune opération sûre sur elles ne peut introduire de l'UTF-8 mal formé. Cela simplifie le code qui fonctionne avec le texte.

Rust place les méthodes de gestion de texte sur `str` ou `String` selon que la méthode nécessite un tampon redimensionnable ou se contente d'utiliser le texte en place. Depuis `String` les déréférences à `&str`, chaque méthode définie sur `str` est également directement disponible sur `String`. Cette section présente les méthodes des deux types, regroupées par fonction approximative.

Ces méthodes indexent le texte par décalages d'octets et mesurent sa longueur en octets plutôt qu'en caractères. En pratique, compte tenu de la nature d'Unicode, l'indexation par caractère n'est pas aussi utile qu'il y paraît, et les décalages d'octets sont plus rapides et plus simples. Si vous essayez d'utiliser un décalage d'octet qui atterrit au milieu de l'encodage UTF-8 d'un caractère, la méthode panique, vous ne pouvez donc pas introduire un UTF-8 mal formé de cette façon.

A `String` est implémenté comme un wrapper autour de a `Vec<u8>` qui garantit que le contenu du vecteur est toujours bien formé en UTF-8. Rust ne changera jamais `String` pour utiliser une représentation plus compliquée, vous pouvez donc supposer que `String` partage `Vec` les caractéristiques de performance de .

Dans ces explications, les variables ont les types indiqués dans le tableau 17-3 .

Tableau 17-3. Types de variables utilisées dans les explications

Variable	Type présumé
`string`	`String`
`slice`	`&str` ou quelque chose qui déréférence à un, comme `String` ou `Rc<String>`
`ch`	`char`
`n`	`usize`, une longueur
`i`, `j`	`usize`, un décalage d'octet
`range`	Une plage de `usize` décalages d'octets, entièrement délimitée comme `i..j`, ou partiellement délimitée comme `i..`, `..j` ou `..`
`pattern`	Tout type de motif : `char`, `String`, `&str`, `&[char]`, ou `FnMut(char) -> bool`

Nous décrivons les types de modèles dans « Modèles de recherche de texte » .

Création de valeurs de chaîne

Il existe quelques façons courantes de créer des `String` valeurs:

`String::new()`

Retourune nouvelle chaîne vide. Cela n'a pas de tampon alloué au tas, mais en allouera un selon les besoins.

`String::with_capacity(n)`

Retourune nouvelle chaîne vide avec un tampon pré-alloué pour contenir au moins des `n` octets. Si vous connaissez à l'avance la longueur de la chaîne que vous construisez, ce constructeur vous permet de dimensionner correctement le tampon dès le début, au lieu de redimensionner le tampon au fur et à mesure que vous construisez la chaîne. La chaîne augmentera toujours son tampon selon les besoins si sa longueur dépasse les `n` octets. Comme les vecteurs, les chaînes ont des méthodes `capacity`, `reserve` et `shrink_to_fit`, mais généralement la logique d'allocation par défaut convient.

`str_slice.to_string()`

Attribueun fichier frais `String` dont le contenu est une copie de `str_slice` . Nous avons utilisé des expressions comme `"literal text".to_string()` tout au long du livre pour créer `String` des s à partir de littéraux de chaîne.

`iter.collect()`

Constructionsune chaîne en concaténant les éléments d'un itérateur, qui peuvent être `char` , `&str` ou `String` des valeurs. Par exemple, pour supprimer tous les espaces d'une chaîne, vous pouvez écrire :

```
let spacey = "man hat tan";
let spaceless:String =
    spacey.chars().filter(|c| !c.is_whitespace()).collect();
assert_eq!(spaceless, "manhattan");
```

L'utilisation `collect` de cette méthode tire parti de `String` l'implémentation du `std::iter::FromIterator` trait par.

`slice.to_owned()`

Retourune copie de la tranche en tant que fichier fraichement alloué `String` . Le `str` type ne peut pas implémenter `Clone` : le trait nécessiterait `clone` sur a `&str` de retourner une `str` valeur, mais `str` n'est pas dimensionné. Cependant, `&str` implemente `ToOwned` , ce qui permet à l'implémenteur de spécifier son équivalent possédé.

Inspection simplifiée

Ces méthodes obtiennent des informations de baseà partir de tranches de chaîne :

`slice.len()`

La duréede `slice` , en octets.

`slice.is_empty()`

Vraisi `slice.len() == 0` .

`slice[range]`

Retourune tranche empruntant la portion donnée de `slice` . Les plages partiellement délimitées et non délimitées sont acceptables ; par exemple:

```
let full = "bookkeeping";
assert_eq!(&full[..4], "book");
assert_eq!(&full[5..], "eeping");
```

```
assert_eq!(&full[2..4], "ok");
assert_eq!(full[..].len(), 11);
assert_eq!(full[5..].contains("boo"), false);
```

Notez que vous ne pouvez pas indexer une tranche de chaîne avec une seule position, comme `slice[i]`. Récupérer un seul caractère à un décalage d'octet donné est un peu maladroit : vous devez produire un `chars` itérateur sur la tranche et lui demander d'analyser l'UTF-8 d'un caractère :

```
let parenthesized = "Rust (饂)";
assert_eq!(parenthesized[6..].chars().next(), Some('饂'));
```

Cependant, vous devriez rarement avoir besoin de le faire. Rust a des façons bien plus agréables d'itérer sur les tranches, que nous décrivons dans "Itérer sur du texte".

`slice.split_at(i)`

Retourun tuple de deux tranches partagées empruntées à `slice` : la partie jusqu'à l'offset d'octet `i` et la partie qui suit. En d'autres termes, cela renvoie `(slice[..i], slice[i..])`.

`slice.is_char_boundary(i)`

Vraisi le décalage d'octet `i` tombe entre les limites de caractère et convient donc comme décalage dans `slice`.

Naturellement, les tranches peuvent être comparées pour l'égalité, ordonnées et hachées. La comparaison ordonnée traite simplement la chaîne comme une séquence de points de code Unicode et les compare dans l'ordre lexicographique.

Ajout et insertion de texte

Les méthodes suivantesajouter du texte à un `String` :

`string.push(ch)`

Ajoutele personnage `ch` jusqu'au bout `string`.

`string.push_str(slice)`

Ajoutele contenu complet de `slice`.

`string.extend(iter)`

Ajouteles éléments produits par l'itérateur `iter` à la chaîne. L'itérateur peut produire des valeurs `char`, `str` ou . `String` Voici

`String` les implémentations de `std::iter::Extend` :

```
let mut also_spaceless = "con".to_string();
also_spaceless.extend("tri but ion".split_whitespace());
assert_eq!(also_spaceless, "contribution");
```

`string.insert(i, ch)`

Encartsle caractère unique `ch` au décalage d'octet `i` dans `string` . Cela implique de déplacer tous les caractères après `i` pour faire de la place pour `ch` , donc la construction d'une chaîne de cette manière peut nécessiter un temps quadratique dans la longueur de la chaîne.

`string.insert_str(i, slice)`

Cettefait de même pour `slice` , avec la même mise en garde en matière de performances.

`String` met en œuvre `std::fmt::Write` , c'est-à-direque les macros `write!` et `writeln!` peut ajouter du texte formaté à `String` s :

```
use std:: fmt::Write;

let mut letter = String::new();
writeln!(letter, "Whose {} these are I think I know", "rutabagas")?;
writeln!(letter, "His house is in the village though;")?;
assert_eq!(letter, "Whose rutabagas these are I think I know\n\
                    His house is in the village though;\n");
```

Puisque `write!` et `writeln!` sont conçus pour écrire dans les flux de sortie, ils renvoient un `Result` , ce que Rust se plaint si vous l'ignorez. Ce code utilise l' `?` opérateur pour le gérer, mais écrire dans a `String` est en fait infaillible, donc dans ce cas, appeler `.unwrap()` serait également OK.

Depuis `String` implémente `Add<&str>` et `AddAssign<&str>` , vous pouvez écrire du code comme ceci :

```
let left = "partners".to_string();
let mut right = "crime".to_string();
assert_eq!(left + " in " + &right, "partners in crime");

right += " doesn't pay";
assert_eq!(right, "crime doesn't pay");
```

Lorsqu'il est appliqué à des chaînes, l' `+` opérateurprend son opérande gauche par valeur, de sorte qu'il peut réellement le réutiliser `String` à la suite de l'addition. Par conséquent, si le tampon de l'opérande de gauche

est suffisamment grand pour contenir le résultat, aucune allocation n'est nécessaire.

Dans un malheureux manque de symétrie, l'opérande gauche de `+` ne peut pas être un `&str`, donc vous ne pouvez pas écrire :

```
let parenthetical = "(" + string + ")";
```

Vous devez plutôt écrire :

```
let parenthetical = "(".to_string() + &string + ")";
```

Cependant, cette restriction décourage la création de chaînes à partir de la fin vers l'arrière. Cette approche fonctionne mal car le texte doit être déplacé à plusieurs reprises vers la fin du tampon.

Construire des chaînes du début à la fin en ajoutant de petits morceaux, cependant, est efficace. A `String` se comporte comme un vecteur, doublant toujours au moins la taille de son tampon lorsqu'il a besoin de plus de capacité. Cela maintient la surcharge de recopie proportionnelle à la taille finale. Même ainsi, utiliser `String::with_capacity` pour créerles chaînes avec la bonne taille de tampon pour commencer évitent tout redimensionnement et peuvent réduire le nombre d'appels à l'allocateur de tas.

Supprimer et remplacer du texte

`String` a quelques méthodespour supprimer du texte (ceux-ci n'affectent pas la capacité de la chaîne ; utilisez -les `shrink_to_fit` si vous avez besoin de libérer de la mémoire) :

`string.clear()`

Réinitialise `string` à la chaîne vide.

`string.truncate(n)`

Rejetstous les caractères après le décalage d'octet `n`, laissant `string` une longueur d'au plus `n`. Si `string` est plus court que `n` octets, cela n'a aucun effet.

`string.pop()`

Supprimele dernier caractère de `string`, le cas échéant, et le renvoie sous la forme d'un `Option<char>`.

`string.remove(i)`

Supprimele caractère à l'octet de décalage `i` de `string` et le renvoie, décalant tous les caractères suivants vers l'avant. Cela prend un temps linéaire dans le nombre de caractères suivants.

`string.drain(range)`

Retourun itérateur sur la plage donnée d'indices d'octets et supprime les caractères une fois l'itérateur supprimé. Les caractères après la plage sont décalés vers l'avant :

```
let mut choco = "chocolate".to_string();
assert_eq!(choco.drain(3..6).collect::<String>(), "col");
assert_eq!(choco, "choate");
```

Si vous souhaitez simplement supprimer la plage, vous pouvez simplement supprimer l'itérateur immédiatement, sans en tirer aucun élément :

```
let mut winston = "Churchill".to_string();
winston.drain(2..6);
assert_eq!(winston, "Chill");
```

`string.replace_range(range, replacement)`

Remplacela plage donnée `string` avec la tranche de chaîne de remplacement donnée. La tranche n'a pas besoin d'être de la même longueur que la plage remplacée, mais à moins que la plage remplacée n'aille jusqu'à la fin de `string`, cela nécessitera de déplacer tous les octets après la fin de la plage :

```
let mut beverage = "a piña colada".to_string();
beverage.replace_range(2..7, "kahlua"); // 'ñ' is two bytes!
assert_eq!(beverage, "a kahlua colada");
```

Conventions de recherche et d'itération

Fonctions de la bibliothèque standard de Rust pour la recherchetexte et itérationsur le texte, suivez certaines conventions de dénomination pour les rendre plus faciles à retenir :

`r`

La plupart des opérations traitent le texte du début à la fin, mais les opérations dont le nom commence par **`r`** fonctionnent de la fin au début. Par exemple, `rsplit` est la version de bout en bout de `split`. Dans certains cas, le changement de direction peut affecter

non seulement l'ordre dans lequel les valeurs sont produites, mais également les valeurs elles-mêmes. Voir le schéma de la Figure 17-3 pour un exemple.

`n`

Les itérateurs dont le nom se termine par **`n`** se limitent à un nombre donné de correspondances.

`_indices`

Les itérateurs dont les noms se terminent par `_indices` produisent, avec leurs valeurs d'itération habituelles, les décalages d'octets dans la tranche à laquelle ils apparaissent.

La bibliothèque standard ne fournit pas toutes les combinaisons pour chaque opération. Par exemple, de nombreuses opérations n'ont pas besoin d'une `n` variante, car il est assez facile de simplement terminer l'itération plus tôt.

Modèles de recherche de texte

Lorsqu'une fonction de bibliothèque standarddoit rechercher, faire correspondre, fractionner ou rogner du texte, il accepte plusieurs types différents pour représenter ce qu'il faut rechercher :

```
let haystack = "One fine day, in the middle of the night";

assert_eq!(haystack.find(','), Some(12));
assert_eq!(haystack.find("night"), Some(35));
assert_eq!(haystack.find(char::is_whitespace), Some(3));
```

Ces types sont appelés *modèles* et la plupart des opérations les prennent en charge :

```
assert_eq!("## Elephants"
           .trim_start_matches(|ch:char| ch == '#' || ch.is_whitespace()),
           "Elephants");
```

La bibliothèque standard prend en charge quatre principaux types de modèles :

- A `char` en tant que modèle correspond à ce caractère.
- Un `String` ou `&str` ou `&&str` en tant que modèle correspond à une sous-chaîne égale au modèle.

- Une `FnMut(char) -> bool` fermeture en tant que modèle correspond à un seul caractère pour lequel la fermeture renvoie vrai.
- A `&[char]` en tant que modèle (pas un `&str`, mais une tranche de `char` valeurs) correspond à n'importe quel caractère unique qui apparaît dans la liste. Notez que si vous écrivez la liste sous la forme d'un tableau littéral, vous devrez peut-être appeler `as_ref()` pour obtenir le bon type :

```
let code = "\t    function noodle() { ";
assert_eq!(code.trim_start_matches([' ', '\t'].as_ref()),
           "function noodle() { ");
// Shorter equivalent: &[' ', '\t'][..]
```

  Sinon, Rust sera confus par le type de tableau de taille fixe `&[char; 2]`, qui n'est malheureusement pas un type de modèle.

Dans le propre code de la bibliothèque, un modèle est tout type qui implémente le `std::str::Pattern` trait. Les détails de `Pattern` ne sont pas encore stables, vous ne pouvez donc pas l'implémenter pour vos propres types dans Rust stable, mais la porte est ouverte pour autoriser les expressions régulières et d'autres modèles sophistiqués à l'avenir. Rust garantit que les types de modèles pris en charge actuellement continueront de fonctionner à l'avenir.

Recherche et remplacement

Rust a quelques méthodes pour rechercher des motifs dans les trancheset éventuellement en les remplaçant par un nouveau texte :

`slice.contains(pattern)`

Retourtrue si `slice` contient une correspondance pour `pattern`.

`slice.starts_with(pattern)`, *`slice.ends_with(pattern)`*

Revenirtrue si `slice` le texte initial ou final correspond à `pattern` :

```
assert!("2017".starts_with(char::is_numeric));
```

`slice.find(pattern)`, *`slice.rfind(pattern)`*

Revenir `Some(i)` si `slice` contient une correspondance pour `pattern`, où `i` est le décalage d'octet auquel le motif apparaît. La `find` méthode renvoie la première correspondance, `rfind` la dernière :

```
let quip = "We also know there are known unknowns";
assert_eq!(quip.find("know"), Some(8));
assert_eq!(quip.rfind("know"), Some(31));
assert_eq!(quip.find("ya know"), None);
assert_eq!(quip.rfind(char::is_uppercase), Some(0));
```

slice.replace(pattern, replacement)

Retourun nouveau `String` formé en remplaçant avec empressement tous les matches pour `pattern` par `replacement` :

```
assert_eq!("The only thing we have to fear is fear itself"
           .replace("fear", "spin"),
           "The only thing we have to spin is spin itself");

assert_eq!("`Borrow` and `BorrowMut`"
           .replace(|ch:char| !ch.is_alphanumeric(), ""),
           "BorrowandBorrowMut");
```

Étant donné que le remplacement est effectué avec empressement, `.replace()` le comportement de lors des correspondances qui se chevauchent peut être surprenant. Ici, il y a quatre instances du modèle, `"aba"`, mais la deuxième et la quatrième ne correspondent plus après le remplacement de la première et de la troisième :

```
assert_eq!("cababababababbage"
           .replace("aba", "***"),
           "c***b***babbage")
```

slice.replacen(pattern, replacement, n)

Cettefait la même chose, mais remplace au maximum les premiers `n` matchs.

Itérer sur du texte

La bibliothèque standard fournit plusieurs façons d'itérersur le texte d'une tranche. La figure 17-3 en montre des exemples.

Vous pouvez considérer les familles `split` et `match` comme étant complémentaires : les fractionnements sont les plages entre les correspondances.

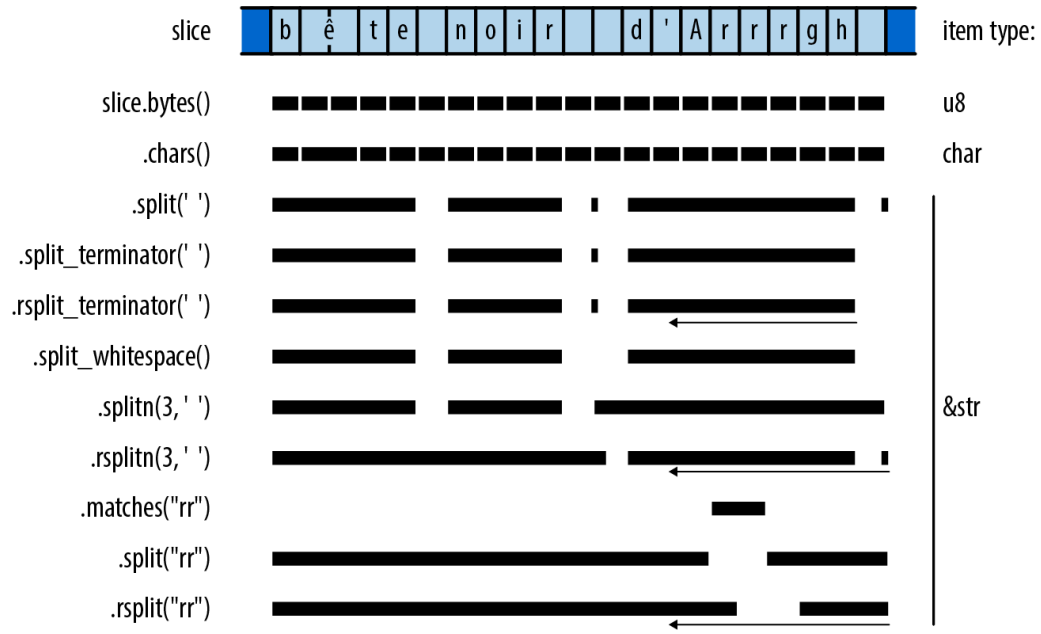

Illustration 17-3. Quelques façons d'itérer sur une tranche

La plupart de ces méthodes renvoient des itérateurs réversibles (c'est-à-dire qu'elles implémentent `DoubleEndedIterator`) : l'appel de leur `.rev()` méthode d'adaptation vous donne un itérateur qui produit les mêmes éléments, mais dans l'ordre inverse.

`slice.chars()`

Retourun itérateur sur `slice` les caractères de .

`slice.char_indices()`

Retourun itérateur sur `slice` les caractères de et leurs décalages d'octet :

```
assert_eq!("élan".char_indices().collect::<Vec<_>>(),
           vec![(0, 'é'), // has a two-byte UTF-8 encoding
                (2, 'l'),
                (3, 'a'),
                (4, 'n')]);
```

Notez que ce n'est pas équivalent à `.chars().enumerate()`, car il fournit le décalage d'octet de chaque caractère dans la tranche, au lieu de simplement numéroter les caractères.

`slice.bytes()`

Retourun itérateur sur les octets individuels de `slice`, exposant l'encodage UTF-8 :

```
assert_eq!("élan".bytes().collect::<Vec<_>>(),
           vec![195, 169, b'l', b'a', b'n']);
```

slice.lines()

Retourun itérateur sur les lignes de `slice` . Les lignes se terminent par `"\n"` ou `"\r\n"` . Chaque pièce produite est un `&str` emprunt à `slice` . Les éléments n'incluent pas les caractères de fin de ligne.

slice.split(pattern)

Retourun itérateur sur les portions de `slice` séparées par des correspondances de `pattern` . Cela produit des chaînes vides entre les correspondances immédiatement adjacentes, ainsi que pour les correspondances au début et à la fin de `slice` .

L'itérateur retourné n'est pas réversible si `pattern` est un `&str` . De tels modèles peuvent produire différentes séquences de correspondances en fonction de la direction à partir de laquelle vous numérisez, ce que les itérateurs réversibles sont interdits de faire. Au lieu de cela, vous pourrez peut-être utiliser la `rsplit` méthode décrite ci-après.

slice.rsplit(pattern)

Cetteest la même, mais analyse `slice` de bout en bout, produisant des correspondances dans cet ordre.

slice.split_terminator(pattern) , *slice.rsplit_terminator (pattern)*

Cessont similaires, sauf que le motif est traité comme un terminateur, pas comme un séparateur : si `pattern` les correspondances sont à la toute fin de `slice` , les itérateurs ne produisent pas de tranche vide représentant la chaîne vide entre cette correspondance et la fin de la tranche, comme `split` et `rsplit` faire . Par exemple:

```
// The ':' characters are separators here. Note the final "".
assert_eq!("jimb:1000:Jim Blandy:".split(':').collect::<Vec<_>>(),
           vec!["jimb", "1000", "Jim Blandy", ""]);

// The '\n' characters are terminators here.
assert_eq!("127.0.0.1  localhost\n\
            127.0.0.1  www.reddit.com\n"
           .split_terminator('\n').collect::<Vec<_>>(),
           vec!["127.0.0.1  localhost",
                "127.0.0.1  www.reddit.com"]);
                // Note, no final ""!
```

slice.splitn(n, pattern) , *slice.rsplitn(n, pattern)*

Cessont comme `split` et `rsplit` , sauf qu'ils divisent la chaîne en au plus `n` tranches, à la première ou à la dernière `n-1` correspondance pour `pattern` .

`slice.split_whitespace()`, `slice.split_ascii_whitespace()`

Revenirun itérateur sur les parties séparées par des espaces de `slice` . Une série de plusieurs caractères d'espacement est considérée comme un seul séparateur. Les espaces de fin sont ignorés.

La `split_whitespace` méthode utilise la définition Unicode des espaces blancs, telle qu'implémentée par la `is_whitespace` méthode sur `char` . La `split_ascii_whitespace` méthode utilise à la `char::is_ascii_whitespace` place, qui ne reconnaît que les caractères d'espacement ASCII.

```
let poem = "This  is  just  to say\n\
            I have eaten\n\
            the plums\n\
            again\n";

assert_eq!(poem.split_whitespace().collect::<Vec<_>>(),
           vec!["This", "is", "just", "to", "say",
                "I", "have", "eaten", "the", "plums",
                "again"]);
```

`slice.matches(pattern)`

Retourun itérateur sur les correspondances pour `pattern` in slice. `slice.rmatches(pattern)` est le même, mais itère de la fin au début.

`slice.match_indices(pattern)`, `slice.rmatch_indices(pattern)`

Cessont similaires, sauf que les éléments produits sont des `(offset, match)` paires, où `offset` est le décalage d'octet auquel la correspondance commence, et `match` est la tranche correspondante.

Garniture

Pour *tailler*une chaîne consiste à supprimer du texte, généralement des espaces, au début ou à la fin de la chaîne. C'est souvent utile pour nettoyer l'entrée lue à partir d'un fichier où l'utilisateur peut avoir mis du texte en retrait pour la lisibilité ou laissé accidentellement un espace blanc à la fin d'une ligne.

`slice.trim()`

Retourune sous-tranche de `slice` qui omet tout espace blanc de début et de fin. `slice.trim_start()` omet uniquement les espaces blancs de début, `slice.trim_end()` uniquement les espaces blancs de fin :

```
assert_eq!("\t*.rs  ".trim(), "*.rs");
assert_eq!("\t*.rs  ".trim_start(), "*.rs  ");
assert_eq!("\t*.rs  ".trim_end(), "\t*.rs");
```

`slice.trim_matches(pattern)`

Retourune sous-tranche de `slice` qui omet toutes les correspondances de `pattern` du début à la fin. Les méthodes `trim_start_matches` et `trim_end_matches` font de même pour les correspondances de début ou de fin uniquement :

```
assert_eq!("001990".trim_start_matches('0'), "1990");
```

`slice.strip_prefix(pattern)`, `slice.strip_suffix(pattern)`

Si `slice` commence par `pattern`, `strip_prefix` retourne `Some` en tenant la tranche avec le texte correspondant supprimé. Sinon, ça revient `None`. La `strip_suffix` méthode est similaire, mais recherche une correspondance à la fin de la chaîne.

Ce sont comme `trim_start_matches` et `trim_end_matches`, sauf qu'ils renvoient un `Option`, et une seule copie de `pattern` est supprimée :

```
let slice = "banana";
assert_eq!(slice.strip_suffix("na"),
           Some("bana"))
```

Conversion de casse pour les chaînes

Les méthodes `slice.to_uppercase()` et le `slice.to_lowercase()` retourune chaîne fraîchement allouéecontenant le texte `slice` converti en majuscule ou en minuscule. Le résultat peut ne pas être de la même longueur que `slice` ; voir "Conversion de la casse pour les caractères" pour plus de détails.

Analyse d'autres types à partir de chaînes

Rust fournit des traits standard pour l'analysevaleurs à partir de chaînes et produire des représentations textuelles des valeurs.

Si un type implémente le `std::str::FromStr` trait, il fournit alors un moyen standard d'analyser une valeur à partir d'une tranche de chaîne :

```
pub trait FromStr: Sized {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

Tous les types de machines usuelles mettent en œuvre `FromStr` :

```
use std:: str::FromStr;

assert_eq!(usize:: from_str("3628800"), Ok(3628800));
assert_eq!(f64:: from_str("128.5625"), Ok(128.5625));
assert_eq!(bool::from_str("true"), Ok(true));

assert!(f64:: from_str("not a float at all").is_err());
assert!(bool::from_str("TRUE").is_err());
```

Le `char` type implémente également `FromStr` , pour les chaînes avec un seul caractère :

```
assert_eq!(char:: from_str("é"), Ok('é'));
assert!(char::from_str("abcdefg").is_err());
```

Le `std::net::IpAddr` type, `enum` qui détient une adresse Internet IPv4 ou IPv6, implémente `FromStr` également :

```
use std:: net::IpAddr;

let address = IpAddr:: from_str("fe80::0000:3ea9:f4ff:fe34:7a50")?;
assert_eq!(address,
           IpAddr::from([0xfe80, 0, 0, 0, 0x3ea9, 0xf4ff, 0xfe34, 0x7a50])
```

Les tranches de chaîne ont une `parse` méthodequi analyse la tranche dans le type de votre choix, en supposant qu'elle implémente `FromStr` . Comme pour `Iterator::collect` , vous aurez parfois besoin d'épeler le type que vous voulez, ce `parse` n'est donc pas toujours beaucoup plus lisible que d'appeler `from_str` directement :

```
let address = "fe80::0000:3ea9:f4ff:fe34:7a50".parse::<IpAddr>()?;
```

Conversion d'autres types en chaînes

Il existe trois façons principales de convertirvaleurs non textuelles en chaînes :

- Les types qui ont une forme imprimée naturelle lisible par l'homme peuvent implémenter le `std::fmt::Display` trait, qui vous permet d'utiliser le `{}` spécificateur de format dans la `format!` macro :

  ```
  assert_eq!(format!("{}, wow", "doge"), "doge, wow");
  assert_eq!(format!("{}", true), "true");
  assert_eq!(format!("({:.3}, {:.3})", 0.5, f64::sqrt(3.0)/2.0),
             "(0.500, 0.866)");

  // Using `address` from above.
  let formatted_addr:String = format!("{}", address);
  assert_eq!(formatted_addr, "fe80::3ea9:f4ff:fe34:7a50");
  ```

  Tous les types numériques de machine de Rust implémentent `Display` , tout comme les caractères, les chaînes et les tranches. Le pointeur intelligent type `Box<T>` , `Rc<T>` , et `Arc<T>` implement `Display` si `T` lui-même le fait : leur forme affichée est simplement celle de leur référent. Les conteneurs aiment `Vec` et `HashMap` n'implémentent pas `Display` , car il n'y a pas de forme naturelle lisible par l'homme pour ces types.
- Si un type implémente `Display` , la bibliothèque standard implémente automatiquement le `std::str::ToString` trait correspondant, dont la seule méthode `to_string` peut être plus pratique lorsque vous n'avez pas besoin de la flexibilité de `format!` :

  ```
  // Continued from above.
  assert_eq!(address.to_string(), "fe80::3ea9:f4ff:fe34:7a50");
  ```

  Le `ToString` trait est antérieur à l'introduction de `Display` et est moins flexible. Pour vos propres types, vous devez généralement implémenter à la `Display` place de `ToString` .
- Toustype public dans la bibliothèque standard implements `std::fmt::Debug` , qui prend une valeur et la formate en tant que chaîne d'une manière utile aux programmeurs. Le moyen le plus simple `Debug` de produire une chaîne consiste à utiliser le spécificateur de format de la `format!` macro : `{:?}`

```
// Continued from above.
let addresses = vec![address,
                     IpAddr::from_str("192.168.0.1")?];
assert_eq!(format!("{:?}", addresses),
           "[fe80::3ea9:f4ff:fe34:7a50, 192.168.0.1]");
```

Cela tire parti d'une implémentation globale de `Debug` for `Vec<T>`, pour tout `T` ce qui implémente lui-même `Debug`. Tous les types de collection de Rust ont de telles implémentations.
Vous devez également implémenter `Debug` pour vos propres types. Habituellement, il est préférable de laisser Rust dériver une implémentation, comme nous l'avons fait pour le `Complex` type au chapitre 12 :

```
#[derive(Copy, Clone, Debug)]
struct Complex { re: f64, im:f64 }
```

Les traits `Display` et de `Debug` formatage ne sont que deux parmi plusieurs que la `format!` macro et ses parents utilisent pour formater les valeurs sous forme de texte. Nous couvrirons les autres, et expliquerons comment les implémenter tous, dans "Formatage des valeurs".

Emprunter sous d'autres types de texte

Tu peux emprunterle contenu d'une tranche de plusieurs manières :

- Trancheset `String`s implémentent `AsRef<str>`, `AsRef<[u8]>`, `AsRef<Path>` et `AsRef<OsStr>`. De nombreuses fonctions de bibliothèque standard utilisent ces traits comme limites sur leurs types de paramètres, vous pouvez donc leur transmettre directement des tranches et des chaînes, même lorsqu'elles veulent vraiment un autre type. Voir « AsRef et AsMut » pour une explication plus détaillée.
- Les tranches et les chaînes implémentent également le `std::borrow::Borrow<str>` trait. `HashMap` et `BTreeMap` utiliser `Borrow` pour que `String`s fonctionne bien comme clés dans une table. Voir "Emprunter et EmprunterMut" pour plus de détails.

Accéder au texte en UTF-8

Il y a deux principauxfaçons d'accéder aux octets représentant le texte, selon que vous souhaitez vous approprier les octets ou simplement les emprunter :

`slice.as_bytes()`

Emprunte `slice` les octets de `&[u8]`. Comme il ne s'agit pas d'une référence mutable, `slice` on peut supposer que ses octets resteront bien formés en UTF-8.

`string.into_bytes()`

S'approprie `string` et renvoie un `Vec<u8>` des octets de la chaîne par valeur. Il s'agit d'une conversion bon marché, car elle transmet simplement le `Vec<u8>` que la chaîne utilisait comme tampon. Comme `string` il n'existe plus, il n'est pas nécessaire que les octets continuent d'être bien formés en UTF-8, et l'appelant est libre de modifier le `Vec<u8>` à sa guise.

Production de texte à partir de données UTF-8

Si vous avez un bloc d'octetsque vous pensez contenir des données UTF-8, vous avez quelques options pour les convertir en `String` s ou tranches, selon la façon dont vous souhaitez gérer les erreurs :

`str::from_utf8(byte_slice)`

Prendune `&[u8]` tranche d'octets et renvoie un `Result` : soit `Ok(&str)` s'il `byte_slice` contient de l'UTF-8 bien formé, soit une erreur dans le cas contraire.

`String::from_utf8(vec)`

Essaiepour construire une chaîne à partir d'une `Vec<u8>` valeur passée par. Si `vec` contient UTF-8 bien formé, `from_utf8` renvoie `Ok(string)`, où `string` a pris possession de `vec` pour l'utiliser comme tampon. Aucune allocation de tas ou copie du texte n'a lieu.

Si les octets ne sont pas UTF-8 valides, cela renvoie `Err(e)`, où `e` est une `FromUtf8Error` valeur d'erreur. L'appel `e.into_bytes()` vous renvoie le vector d'origine `vec`, de sorte qu'il n'est pas perdu lorsque la conversion échoue :

```
let good_utf8: Vec<u8> = vec![0xe9, 0x8c, 0x86];
assert_eq!(String::from_utf8(good_utf8).ok(), Some("錆".to_string()));

let bad_utf8:   Vec<u8> = vec![0x9f, 0xf0, 0xa6, 0x80];
let result = String::from_utf8(bad_utf8);
assert!(result.is_err());
// Since String::from_utf8 failed, it didn't consume the original
// vector, and the error value hands it back to us unharmed.
assert_eq!(result.unwrap_err().into_bytes(),
           vec![0x9f, 0xf0, 0xa6, 0x80]);
```

`String::from_utf8_lossy(byte_slice)`

Essaiepour construire un `String` ou `&str` à partir d'une `&[u8]` tranche partagée d'octets. Cette conversion réussit toujours, remplaçant tout UTF-8 mal formé par des caractères de remplacement Unicode. La valeur de retour est a `Cow<str>` qui emprunte `&str` directement a `byte_slice` s'il contient de l'UTF-8 bien formé ou possède un nouvellement alloué `String` avec des caractères de remplacement substitués aux octets mal formés. Par conséquent, lorsque `byte_slice` est bien formé, aucune allocation de tas ou copie n'a lieu. Nous en discuterons `Cow<str>` plus en détail dans "Reporter l'allocation" .

`String::from_utf8_unchecked(vec)`

Si vous savez pertinemmentque votre `Vec<u8>` contient UTF-8 bien formé, vous pouvez appeler cette fonction non sécurisée. Cela se termine simplement `vec` par un `String` et le renvoie, sans examiner du tout les octets. Vous êtes responsable de vous assurer que vous n'avez pas introduit d'UTF-8 mal formé dans le système, c'est pourquoi cette fonction est marquée `unsafe` .

`str::from_utf8_unchecked(byte_slice)`

De la même manière, cela prend a `&[u8]` et le renvoie comme a `&str` , sans vérifier s'il contient de l'UTF-8 bien formé. Comme pour `String::from_utf8_unchecked` , vous êtes responsable de vous assurer que cela est sûr.

Différer l'attribution

Supposervous voulez que votre programme accueille l'utilisateur. Sous Unix, vous pourriez écrire :

```
fn get_name() -> String {
    std:: env::var("USER") // Windows uses "USERNAME"
        .unwrap_or("whoever you are".to_string())
}

println!("Greetings, {}!", get_name());
```

Pour les utilisateurs Unix, cela les accueille par nom d'utilisateur. Pour les utilisateurs de Windows et les tragiquement anonymes, il fournit un texte de stock alternatif.

La `std::env::var` fonction renvoie un `String` —et a de bonnes raisons de le faire que nous n'aborderons pas ici. Mais cela signifie que le texte de stock alternatif doit également être renvoyé en tant que fichier `String` . C'est décevant : lors `get_name` du retour d'une chaîne statique, aucune allocation ne devrait être nécessaire du tout.

Le nœud du problème est que parfois la valeur de retour de `get_name` devrait être un Owned `String`, parfois ce devrait être un `&'static str`, et nous ne pouvons pas savoir lequel ce sera tant que nous n'aurons pas exécuté le programme. Ce caractère dynamique est l'indice à envisager d'utiliser `std::borrow::Cow`, le clone-type en écriture pouvant contenir des données détenues ou empruntées.

Comme expliqué dans "Borrow and ToOwned at Work: The Humble Cow", `Cow<'a, T>` est une énumération avec deux variantes : `Owned` et `Borrowed`. `Borrowed` contient une référence `&'a T` et `Owned` contient la version propriétaire de `&T` : `String` for `&str`, `Vec<i32>` for `&[i32]`, etc. Que ce soit `Owned` ou `Borrowed`, a `Cow<'a, T>` peut toujours produire un `&T` pour que vous puissiez l'utiliser. En fait, `Cow<'a, T>` les déréférences à `&T`, se comportent comme une sorte de pointeur intelligent.

Changer `get_name` pour renvoyer un `Cow` résultat dans ce qui suit :

```
use std:: borrow::Cow;

fn get_name() -> Cow<'static, str> {
    std:: env:: var("USER")
        .map(|v| Cow:: Owned(v))
        .unwrap_or(Cow::Borrowed("whoever you are"))
}
```

Si cela réussit à lire la `"USER"` variable d'environnement, le `map` renvoie le résultat `String` sous la forme d'un fichier `Cow::Owned`. En cas d'échec, le `unwrap_or` renvoie son statique `&str` sous la forme d'un fichier `Cow::Borrowed`. L'appelant peut rester inchangé :

```
println!("Greetings, {}!", get_name());
```

Tant que `T` le `std::fmt::Display` trait est implémenté, l'affichage d'un `Cow<'a, T>` produit les mêmes résultats que l'affichage d'un `T`.

`Cow` est également utile lorsque vous pouvez ou non avoir besoin de modifier un texte que vous avez emprunté. Lorsqu'aucune modification n'est nécessaire, vous pouvez continuer à l'emprunter. Mais le comportement de clonage sur écriture homonyme de `Cow` peut vous donner une copie propriétaire et modifiable de la valeur à la demande. `Cow` La `to_mut` méthode de s'assure que le `Cow` est `Cow::Owned`, en appliquant l' `ToOwned` implémentation de la valeur si nécessaire, puis renvoie une référence mutable à la valeur.

Ainsi, si vous constatez que certains de vos utilisateurs, mais pas tous, ont des titres par lesquels ils préféreraient être adressés, vous pouvez dire :

```
fn get_title() ->Option<&'static str> { ... }

let mut name = get_name();
if let Some(title) = get_title() {
    name.to_mut().push_str(", ");
    name.to_mut().push_str(title);
}

println!("Greetings, {}!", name);
```

Cela peut produire une sortie comme celle-ci :

```
$course de fret
Greetings, jimb, Esq.!
$
```

Ce qui est bien ici, c'est que si `get_name()` renvoie une chaîne statique et `get_title` renvoie `None`, le `Cow` transporte simplement la chaîne statique jusqu'au `println!`. Vous avez réussi à reporter l'allocation à moins que ce ne soit vraiment nécessaire, tout en écrivant du code simple.

Étant donné `Cow` qu'il est fréquemment utilisé pour les chaînes, la bibliothèque standard a un support spécial pour `Cow<'a, str>`. Il fournit `From` et `Into` convertit à la fois `String` et `&str`, vous pouvez donc écrire `get_name` plus brièvement :

```
fn get_name() -> Cow<'static, str> {
    std:: env::var("USER")
        .map(|v| v.into())
        .unwrap_or("whoever you are".into())
}
```

`Cow<'a, str>` implémente également `std::ops::Add` et `std::ops::AddAssign`, donc pour ajouter le titre au nom, vous pouvez écrire :

```
if let Some(title) = get_title() {
    name += ", ";
    name += title;
}
```

Ou, puisque a `String` peut être `write!` la destination d'une macro :

```
use std:: fmt::Write;

if let Some(title) = get_title() {
    write!(name.to_mut(), ", {}", title).unwrap();
}
```

Comme précédemment, aucune allocation ne se produit tant que vous n'essayez pas de modifier le fichier `Cow` .

Gardez à l'esprit que tous ne `Cow<..., str>` doivent pas l'être `'static` : vous pouvez utiliser `Cow` pour emprunter du texte précédemment calculé jusqu'au moment où une copie devient nécessaire.

Chaînes en tant que collections génériques

`String` met en oeuvreles deux `std::default::Default` et `std::iter::Extend` : `default` renvoie une chaîne vide et `extend` peut ajouter des caractères, des tranches de chaîne, des `Cow<..., str>` s ou des chaînes à la fin d'une chaîne. Il s'agit de la même combinaison de traits implémentés par les autres types de collection de Rust comme `Vec` et `HashMap` pour les modèles de construction génériques tels que `collect` et `partition` .

Le `&str` type implémente également `Default` , renvoyant une tranche vide. C'est pratique dans certains cas d'angle ; par exemple, il vous permet de dériver `Default` des structures contenant des tranches de chaîne.

Valeurs de formatage

Tout au long du livre, nous avons utilisé la mise en forme du textedes macros comme `println!` :

```
println!("{:.3}µs: relocated {} at {:#x} to {:#x}, {} bytes",
         0.84391, "object",
         140737488346304_usize, 6299664_usize, 64);
```

Cet appel produit la sortie suivante :

```
0.844µs: relocated object at 0x7fffffffdcc0 to 0x602010, 64 bytes
```

Le littéral de chaîne sert de modèle pour la sortie : chacun `{...}` dans le modèle est remplacé par la forme formatée de l'un des arguments suivants. La chaîne de modèle doit être une constante afin que Rust puisse la comparer aux types des arguments au moment de la compilation. Chaque argument doit être utilisé ; Sinon, Rust signale une erreur de compilation.

Plusieurs fonctionnalités de la bibliothèque standard partagent ce petit langage de formatage des chaînes :

- La `format!` macrol'utilise pour construire `String` s.
- Les macros `println!` et `print!` écrire du texte formaté dans le flux de sortie standard.
- Les macros `writeln!` et `write!` l'écrire dans un flux de sortie désigné.
- La `panic!` macrol'utilise pour construire une expression (idéalement informative) de consternation terminale.

Les fonctions de formatage de Rust sont conçues pour être ouvertes. Vous pouvez étendre ces macros pour prendre en charge vos propres types en implémentant les `std::fmt` traits de formatage du module. Et vous pouvez utiliser la `format_args!` macroet le `std::fmt::Arguments` type pour que vos propres fonctions et macros prennent en charge le langage de formatage.

Les macros de formatage empruntent toujours des références partagées à leurs arguments ; ils ne s'en approprient jamais ni ne les transforment.

Les formulaires du modèle `{...}` sont appelés *paramètres de format*et avoir le formulaire . Les deux parties sont facultatives ; est fréquemment utilisé. `{which:how} {}`

La *`which`* valeur sélectionne l'argument suivant le modèle qui doit prendre la place du paramètre. Vous pouvez sélectionner des arguments par index ou par nom. Les paramètres sans *`which`* valeur sont simplement associés à des arguments de gauche à droite.

La *`how`* valeur indique comment l'argument doit être formaté : combien de rembourrage, à quelle précision, dans quelle base numérique, etc. Si *`how`* est présent, le côlon avant est requis. Le tableau 17-4 présente quelques exemples.

Tableau 17-4. Exemples de chaînes formatées

Chaîne de modèle	Liste des arguments	Résultat
`"number of {}: {}"`	`"elephants", 19`	`"number of elephants: 19"`
`"from {1} to {0}"`	`"the grave", "the cradle"`	`"from the cradle to the grave"`
`"v = {:?}"`	`vec![0,1,2,5,12,29]`	`"v = [0, 1, 2, 5, 12, 29]"`
`"name = {:?}"`	`"Nemo"`	`"name = \"Nemo\""`
`"{:8.2} km/s"`	`11.186`	`"   11.19 km/s"`
`"{:20} {:02x} {:02x}"`	`"adc #42", 105, 42`	`"adc #42 69 2a"`
`"{1:02x} {2:02x}  {0}"`	`"adc #42", 105, 42`	`"69 2a  adc #42"`
`"{lsb:02x} {msb:02x}  {insn}"`	`insn="adc #42", lsb=105, msb=42`	`"69 2a  adc #42"`
`"{:02?}"`	`[110, 11, 9]`	`"[110, 11, 09]"`
`"{:02x?}"`	`[110, 11, 9]`	`"[6e, 0b, 09]"`

Si vous souhaitez inclure des caractères `{` ou `}` dans votre sortie, doublez les caractères dans le modèle :

```
assert_eq!(format!("{{a, c}} ⊂ {{a, b, c}}"),
           "{a, c} ⊂ {a, b, c}");
```

Formatage des valeurs de texte

Lors de la mise en forme d'un textede type comme `&str` ou `String` (`char` est traité comme une chaîne à un seul caractère), la *how* valeur d'un paramètre comporte plusieurs parties, toutes facultatives :

- Une *limite de longueur de texte* . Rust tronque votre argument s'il est plus long que cela. Si vous ne spécifiez aucune limite, Rust utilise le texte intégral.
- Une *largeur de champ minimale* . Après toute troncature, si votre argument est plus court que cela, Rust le remplit à droite (par défaut) avec des espaces (par défaut) pour créer un champ de cette largeur. S'il est omis, Rust ne complète pas votre argument.
- Un *alignement* . Si votre argument doit être rempli pour respecter la largeur de champ minimale, cela indique où votre texte doit être placé dans le champ. `<` , `^` et `>` placez votre texte au début, au milieu et à la fin, respectivement.
- Un caractère de *remplissage* à utiliser dans ce processus de remplissage. S'il est omis, Rust utilise des espaces. Si vous spécifiez le caractère de remplissage, vous devez également spécifier l'alignement.

Le tableau 17-5 illustre quelques exemples montrant comment écrire les choses et leurs effets. Tous utilisent le même argument à huit caractères, `"bookends"` .

Tableau 17-5. Formater les directives de chaîne pour le texte

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Défaut	`"{}"`	`"bookends"`
Largeur de champ minimale	`"{:4}"`	`"bookends"`
	`"{:12}"`	`"bookends "`
Limite de longueur de texte	`"{:.4}"`	`"book"`
	`"{:.12}"`	`"bookends"`
Largeur de champ, limite de longueur	`"{:12.20}"`	`"bookends "`
	`"{:4.20}"`	`"bookends"`
	`"{:4.6}"`	`"booken"`
	`"{:6.4}"`	`"book  "`
Aligné à gauche, largeur	`"{:<12}"`	`"bookends "`
Centré, largeur	`"{:^12}"`	`" bookends "`
Aligné à droite, largeur	`"{:>12}"`	`"  bookends"`
Pad avec `'='`, centré, largeur	`"{:=^12}"`	`"==bookends=="`
Pad `'*'`, aligné à droite, largeur, limite	`"{:*>12.4}"`	`"********book"`

Le formateur de Rust a une compréhension naïve de la largeur : il suppose que chaque caractère occupe une colonne, sans tenir compte des combinaisons de caractères, des katakana demi-largeur, des espaces de

largeur nulle ou des autres réalités désordonnées d'Unicode. Par exemple:

```
assert_eq!(format!("{:4}", "th\u{e9}"),    "th\u{e9} ");
assert_eq!(format!("{:4}", "the\u{301}"), "the\u{301}");
```

Bien qu'Unicode indique que ces chaînes sont toutes deux équivalentes à `"thé"`, le formateur de Rust ne sait pas que des caractères tels que `'\u{301}'`, COMBINING ACUTE ACCENT, nécessitent un traitement spécial. Il remplit correctement la première chaîne, mais suppose que la seconde a une largeur de quatre colonnes et n'ajoute aucun remplissage. Bien qu'il soit facile de voir comment Rust pourrait s'améliorer dans ce cas spécifique, le véritable formatage de texte multilingue pour tous les scripts Unicode est une tâche monumentale, mieux gérée en s'appuyant sur les boîtes à outils de l'interface utilisateur de votre plate-forme, ou peut-être en générant du HTML et du CSS et en créant un site Web. navigateur trier tout cela. Il existe une caisse populaire, `unicode-width`, qui gère certains aspects de cela.

Avec `&str` et `String`, vous pouvez également passer des types de pointeurs intelligents de macros de mise en forme avec des référents textuels, comme `Rc<String>` ou `Cow<'a, str>`, sans cérémonie.

Étant donné que les chemins de nom de fichier ne sont pas nécessairement UTF bien formés-8, `std::path::Path` n'est pas tout à fait un type textuel ; vous ne pouvez pas passer a `std::path::Path` directement à une macro de formatage. Cependant, une `Path` méthode `display` renvoie une valeur que vous pouvez formater et qui trie les choses d'une manière adaptée à la plate-forme:

```
println!("processing file: {}", path.display());
```

Formatage des nombres

Lorsque l'argument de formatage a un nombretapez comme `usize` ou `f64`, la *how* valeur du paramètre comporte les parties suivantes, toutes facultatives :

- Un *rembourrage* et un *alignement*, qui fonctionnent comme ils le font avec les types textuels.
- Un `+` caractère, demandant que le signe du nombre soit toujours affiché, même lorsque l'argument est positif.

- Un `#` caractère, demandant un préfixe de base explicite comme `0x` ou `0b`. Voir la puce "notation" qui conclut cette liste.
- Un `0` caractère, demandant que la largeur de champ minimale soit satisfaite en incluant des zéros non significatifs dans le nombre, au lieu de l'approche de remplissage habituelle.
- Une *largeur de champ minimale*. Si le nombre formaté n'est pas au moins aussi large, Rust le remplit à gauche (par défaut) avec des espaces (par défaut) pour créer un champ de la largeur donnée.
- Une *précision* pour les arguments à virgule flottante, indiquant le nombre de chiffres que Rust doit inclure après la virgule décimale. Rust arrondit ou étend zéro si nécessaire pour produire exactement ce nombre de chiffres fractionnaires. Si la précision est omise, Rust essaie de représenter avec précision la valeur en utilisant le moins de chiffres possible. Pour les arguments de type entier, la précision est ignorée.
- Une *notation*. Pour entiertypes, cela peut être `b` pour les binaires, `o` pour les octaux `x` ou `X` pour les hexadécimaux avec des lettres minuscules ou majuscules. Si vous avez inclus le `#` caractère, ceux-ci incluent un préfixe de base explicite de style Rust `0b`, `0o`, `0x`, ou `0X`. Pour les types à virgule flottante, une base de `e` ou `E` demande une notation scientifique, avec un coefficient normalisé, en utilisant `e` ou `E` pour l'exposant. Si vous ne spécifiez aucune notation, Rust formate les nombres en décimal.

Le tableau 17-6 montre quelques exemples de formatage de la `i32` valeur `1234`.

Tableau 17-6. Formater les directives de chaîne pour les entiers

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Défaut	`"{}"`	`"1234"`
Signe forcé	`"{:+}"`	`"+1234"`
Largeur de champ minimale	`"{:12}"`	`"        1234"`
	`"{:2}"`	`"1234"`
Signe, largeur	`"{:+12}"`	`"       +1234"`
Zéros non significatifs, largeur	`"{:012}"`	`"000000001234"`
Signe, zéros, largeur	`"{:+012}"`	`"+000000001234"`
Aligné à gauche, largeur	`"{:<12}"`	`"1234        "`
Centré, largeur	`"{:^12}"`	`"    1234    "`
Aligné à droite, largeur	`"{:>12}"`	`"        1234"`
Aligné à gauche, signe, largeur	`"{:<+12}"`	`"+1234       "`
Centré, signe, largeur	`"{:^+12}"`	`"   +1234    "`
Aligné à droite, signe, largeur	`"{:>+12}"`	`"       +1234"`
Rembourré avec `'='`, centré, largeur	`"{:=^12}"`	`"====1234====="`

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Notation binaire	`"{:b}"`	`"10011010010"`
Largeur, notation octale	`"{:12o}"`	`"        2322"`
Signe, largeur, notation hexadécimale	`"{:+12x}"`	`"        +4d2"`
Signe, largeur, hexadécimal avec chiffres majuscules	`"{:+12X}"`	`"        +4D2"`
Signe, préfixe de base explicite, largeur, hexadécimal	`"{:+#12x}"`	`"      +0x4d2"`
Signe, base, zéros, largeur, hexadécimal	`"{:+#012x}"`	`"+0x000004d2"`
	`"{:+#06x}"`	`"+0x4d2"`

Comme le montrent les deux derniers exemples, la largeur de champ minimale s'applique au nombre entier, au signe, au préfixe de base et à tous.

Les nombres négatifs incluent toujours leur signe. Les résultats sont similaires à ceux présentés dans les exemples de « signe forcé ».

Lorsque vous demandez des zéros non significatifs, les caractères d'alignement et de remplissage sont simplement ignorés, car les zéros étendent le nombre pour remplir tout le champ.

En utilisant l'argument `1234.5678`, nous pouvons montrer des effets spécifiques à la virgule flottanteles types(Tableau 17-7).

Tableau 17-7. Formater les directives de chaîne pour les nombres à virgule flottante

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Défaut	`"{}"`	`"1234.5678"`
Précision	`"{:.2}"`	`"1234.57"`
	`"{:.6}"`	`"1234.567800"`
Largeur de champ minimale	`"{:12}"`	`"   1234.5678"`
Minimum, précision	`"{:12.2}"`	`"     1234.57"`
	`"{:12.6}"`	`" 1234.567800"`
Zéros non significatifs, minimum, précision	`"{:012.6}"`	`"01234.567800"`
Scientifique	`"{:e}"`	`"1.2345678e3"`
Scientifique, précision	`"{:.3e}"`	`"1.235e3"`
Scientifique, minimum, précision	`"{:12.3e}"`	`"     1.235e3"`
	`"{:12.3E}"`	`"     1.235E3"`

Formatage d'autres types

Au-delà des chaînes et des nombres, vous pouvez formater plusieurs autres types de bibliothèques standard:

- Erreurles types peuvent tous être formatés directement, ce qui facilite leur inclusion dans les messages d'erreur. Chaque type d'erreur doit

implémenter le `std::error::Error` trait, qui étend le trait de formatage par défaut `std::fmt::Display`. Par conséquent, tout type qui implémente `Error` est prêt à être formaté.
- Vous pouvez formater le protocole Internetles types d'adresse comme `std::net::IpAddr` et `std::net::SocketAddr`.
- LaLes booléens `true` et les `false` valeurs peuvent être formatés, bien que ce ne soient généralement pas les meilleures chaînes à présenter directement aux utilisateurs finaux.

Vous devez utiliser les mêmes sortes de paramètres de format que vous utiliseriez pour les chaînes. Les contrôles de limite de longueur, de largeur de champ et d'alignement fonctionnent comme prévu.

Formatage des valeurs pour le débogage

Pour aider au débogageet journalisation, le `{:?}` paramètreformate tout type public dans la bibliothèque standard Rust d'une manière destinée à être utile aux programmeurs. Vous pouvez l'utiliser pour inspecter des vecteurs, des tranches, des tuples, des tables de hachage, des threads et des centaines d'autres types.

Par exemple, vous pouvez écrire ce qui suit :

```
use std:: collections:: HashMap;
let mut map = HashMap::new();
map.insert("Portland", (45.5237606,-122.6819273));
map.insert("Taipei",   (25.0375167, 121.5637));
println!("{:?}", map);
```

Cela imprime :

```
{"Taipei": (25.0375167, 121.5637), "Portland": (45.5237606, -122.6819273)}
```

Les types `HashMap` et `(f64, f64)` savent déjà comment se formater, sans aucun effort de votre part.

Si vous incluez le `#` caractère dans le paramètre de format, Rust imprimera joliment la valeur. Changer ce code pour dire `println!("{:#?}", map)` conduit à cette sortie :

```
{
    "Taipei": (
        25.0375167,
        121.5637
```

```
        ),
        "Portland": (
            45.5237606,
            -122.6819273
        )
    }
```

Ces formes exactes ne sont pas garanties et changent parfois d'une version de Rust à l'autre.

Le formatage de débogage imprime généralement les nombres en décimal, mais vous pouvez mettre un `x` ou `X` avant le point d'interrogation pour demander une valeur hexadécimale à la place. La syntaxe du zéro non significatif et de la largeur de champ est également respectée. Par exemple, vous pouvez écrire :

```
println!("ordinary: {:02?}",  [9, 15, 240]);
println!("hex:      {:02x?}", [9, 15, 240]);
```

Cela imprime :

```
ordinary: [09, 15, 240]
hex:      [09, 0f, f0]
```

Comme nous l'avons mentionné, vous pouvez utiliser la `#[derive(Debug)]` syntaxe pour faire fonctionner vos propres types avec `{:?}` :

```
#[derive(Copy, Clone, Debug)]
struct Complex { re: f64, im:f64 }
```

Avec cette définition en place, nous pouvons utiliser un `{:?}` format pour imprimer les `Complex` valeurs :

```
let third = Complex { re: -0.5, im: f64::sqrt(0.75) };
println!("{:?}", third);
```

Cela imprime :

```
Complex { re: -0.5, im: 0.8660254037844386 }
```

C'est bien pour le débogage, mais ce serait bien si `{}` on pouvait les imprimer sous une forme plus traditionnelle, comme `-0.5 + 0.8660254037844386i` . Dans "Formater vos propres types" , nous montrerons comment faire exactement cela.

Formatage des pointeurs pour le débogage

Normalement, si vous passez n'importe quel type de pointeur à une macro de formatage — une référence, un `Box` , un `Rc` — la macro suit simplement le pointeur et formate son référent ; le pointeur lui-même n'a pas d'intérêt. Mais lorsque vous déboguez, il est parfois utile de voir le pointeur : une adresse peut servir de "nom" approximatif pour une valeur individuelle, ce qui peut être éclairant lors de l'examen de structures avec des cycles ou du partage.

La `{:p}` notationformate les références, les boîtes et autres types de type pointeur en tant qu'adresses :

```
use std:: rc:: Rc;
let original = Rc:: new("mazurka".to_string());
let cloned = original.clone();
let impostor = Rc::new("mazurka".to_string());
println!("text:     {}, {}, {}",       original, cloned, impostor);
println!("pointers: {:p}, {:p}, {:p}", original, cloned, impostor);
```

Ce code imprime :

```
text:     mazurka, mazurka, mazurka
pointers: 0x7f99af80e000, 0x7f99af80e000, 0x7f99af80e030
```

Bien sûr, les valeurs de pointeur spécifiques varient d'une exécution à l'autre, mais même ainsi, la comparaison des adresses montre clairement que les deux premières sont des références au même `String` , tandis que la troisième pointe vers une valeur distincte.

Les adresses ont tendance à ressembler à de la soupe hexadécimale, donc des visualisations plus raffinées peuvent être utiles, mais le `{:p}` style peut toujours être une solution efficace et rapide.

Faire référence aux arguments par index ou par nom

Un paramètre de format peut sélectionner explicitement quel argumentil utilise. Par exemple:

```
assert_eq!(format!("{1},{0},{2}", "zeroth", "first", "second"),
           "first,zeroth,second");
```

Vous pouvez inclure des paramètres de format après deux-points :

```
assert_eq!(format!("{2:#06x},{1:b},{0:=>10}", "first", 10, 100),
           "0x0064,1010,=====first");
```

Vous pouvez également sélectionner des arguments par leur nom. Cela rend les modèles complexes avec de nombreux paramètres beaucoup plus lisibles. Par exemple:

```
assert_eq!(format!("{description:.<25}{quantity:2} @ {price:5.2}",
                   price=3.25,
                   quantity=3,
                   description="Maple Turmeric Latte"),
           "Maple Turmeric Latte..... 3 @  3.25");
```

(Les arguments nommés ici ressemblent aux arguments de mots-clés en Python, mais il ne s'agit que d'une fonctionnalité spéciale des macros de formatage, qui ne fait pas partie de la syntaxe d'appel de fonction de Rust.)

Vous pouvez mélanger des paramètres indexés, nommés et positionnels (c'est-à-dire sans index ni nom) dans une seule macro de mise en forme. Les paramètres positionnels sont associés à des arguments de gauche à droite comme si les paramètres indexés et nommés n'étaient pas là :

```
assert_eq!(format!("{mode} {2} {} {}",
                   "people", "eater", "purple", mode="flying"),
           "flying purple people eater");
```

Les arguments nommés doivent apparaître à la fin de la liste.

Largeurs et précisions dynamiques

Un paramètrechamp minimalla largeur, la limite de longueur du texte et la précision numérique ne doivent pas toujours être des valeurs fixes ; vous pouvez les choisir au moment de l'exécution.

Nous avons examiné des cas comme cette expression, qui vous donne la chaîne `content` justifiée à droite dans un champ de 20 caractères :

```
format!("{:>20}", content)
```

Mais si vous souhaitez choisir la largeur du champ au moment de l'exécution, vous pouvez écrire :

```
format!("{:>1$}", content, get_width())
```

L'écriture `1$` pour la largeur de champ minimale indique `format!` d'utiliser la valeur du deuxième argument comme largeur. L'argument cité doit être un `usize`. Vous pouvez également faire référence à l'argument par son nom :

```
format!("{:>width$}", content, width=get_width())
```

La même approche fonctionne également pour la limite de longueur du texte :

```
format!("{:>width$.limit$}", content,
        width=get_width(), limit=get_limit())
```

À la place de la limite de longueur du texte ou de la précision en virgule flottante, vous pouvez également écrire `*`, qui indique de prendre le prochain argument positionnel comme précision. Les clips suivants `content` à la plupart `get_limit()` des personnages :

```
format!("{:.*}", get_limit(), content)
```

L'argument pris comme précision doit être un `usize`. Il n'y a pas de syntaxe correspondante pour la largeur de champ.

Formater vos propres types

Les macros de formatage utilisent un ensemble de traits définis dans le `std::fmt` modulepour convertir des valeurs en texte. Vous pouvez faire en sorte que les macros de formatage de Rust formatent vos propres types en implémentantun ou plusieurs de ces traits vous-même.

La notationd'un paramètre de format indique quel trait le type de son argument doit implémenter, comme illustré dans le Tableau 17-8 .

Tableau 17-8. Notation de directive de chaîne de format

Notation	Exemple	Caractéristique	Objectif
rien	`{}`	`std::fmt::Display`	Texte, chiffres, erreurs : le trait fourre-tout
`b`	`{bits:#b}`	`std::fmt::Binary`	Nombres en binaire
`o`	`{:#5o}`	`std::fmt::Octal`	Nombres en octal
`x`	`{:4x}`	`std::fmt::LowerHex`	Nombres en hexadécimal, chiffres minuscules
`X`	`{:016X}`	`std::fmt::UpperHex`	Nombres en hexadécimal, chiffres majuscules
`e`	`{:.3e}`	`std::fmt::LowerExp`	Nombres à virgule flottante en notation scientifique
`E`	`{:.3E}`	`std::fmt::UpperExp`	Idem, majuscule `E`
`?`	`{:#?}`	`std::fmt::Debug`	Vue de débogage, pour les développeurs
`p`	`{:p}`	`std::fmt::Pointer`	Pointeur comme adresse, pour les développeurs

Lorsque vous placez l' `#[derive(Debug)]` attribut sur une définition de type afin de pouvoir utiliser le `{:?}` paramètre de format, vous demandez simplement à Rust d'implémenter le `std::fmt::Debug` trait pour vous.

Les traits de formatage ont tous la même structure, ne différant que par leurs noms. Nous utiliserons `std::fmt::Display` comme représentant :

```
trait Display {
    fn fmt(&self, dest: &mut std:: fmt:: Formatter)
        -> std:: fmt::Result;
}
```

Le `fmt` travail de la méthode consiste à produire une représentation correctement formatée de `self` et à écrire ses caractères dans `dest`. En plus de servir de flux de sortie, l' `dest` argument contient également des détails analysés à partir du paramètre de format, comme l'alignement et la largeur de champ minimale.

Par exemple, plus tôt dans ce chapitre, nous avons suggéré qu'il serait bien que `Complex` les valeurs s'impriment elles-mêmes sous la `a + bi` forme habituelle. Voici une `Display` implémentation qui fait cela :

```
use std::fmt;

impl fmt:: Display for Complex {
    fn fmt(&self, dest: &mut fmt:: Formatter) -> fmt:: Result {
        let im_sign = if self.im < 0.0 { '-' } else { '+' };
        write!(dest, "{} {} {}i", self.re, im_sign, f64::abs(self.im))
    }
}
```

Cela tire parti du fait qu'il `Formatter` s'agit lui-même d'un flux de sortie, de sorte que la `write!` macro peut faire la majeure partie du travail pour nous. Avec cette implémentation en place, nous pouvons écrire ce qui suit :

```
let one_twenty = Complex { re: -0.5, im:0.866 };
assert_eq!(format!("{}", one_twenty),
           "-0.5 + 0.866i");

let two_forty = Complex { re: -0.5, im:-0.866 };
assert_eq!(format!("{}", two_forty),
           "-0.5 - 0.866i");
```

Il est parfois utile d'afficher les nombres complexes sous forme polaire : si vous imaginez une ligne tracée sur le plan complexe de l'origine au nombre, la forme polaire donne la longueur de la ligne et son angle dans le sens des aiguilles d'une montre par rapport à l'axe des x positif. Le

`#` caractère dans un paramètre de format sélectionne généralement une autre forme d'affichage ; l' `Display` implémentation pourrait le traiter comme une demande d'utilisation de la forme polaire :

```
impl fmt:: Display for Complex {
    fn fmt(&self, dest: &mut fmt:: Formatter) -> fmt:: Result {
        let (re, im) = (self.re, self.im);
        if dest.alternate() {
            let abs = f64:: sqrt(re * re + im * im);
            let angle = f64:: atan2(im, re) / std:: f64:: consts:: PI * 18
            write!(dest, "{} ∠ {}°", abs, angle)
        } else {
            let im_sign = if im < 0.0 { '-' } else { '+' };
            write!(dest, "{} {} {}i", re, im_sign, f64::abs(im))
        }
    }
}
```

En utilisant cette implémentation :

```
let ninety = Complex { re: 0.0, im:2.0 };
assert_eq!(format!("{}", ninety),
           "0 + 2i");
assert_eq!(format!("{:#}", ninety),
           "2 ∠ 90°");
```

Bien que les méthodes des traits de formatage `fmt` renvoient une `fmt::Result` valeur (un type typique spécifique au module `Result`), vous ne devez propager les échecs qu'à partir des opérations sur le `Formatter` , comme le `fmt::Display` fait l'implémentation avec ses appels à `write!` ; vos fonctions de formatage ne doivent jamais générer elles-mêmes des erreurs. Cela permet aux macros `format!` de renvoyer simplement a `String` au lieu de a `Result<String, ...>` , car l'ajout du texte formaté à a `String` n'échoue jamais. Cela garantit également que toutes les erreurs que vous obtenez `write!` ou `writeln!` reflètent de vrais problèmes du flux d'E / S sous-jacent, et non des problèmes de formatage.

`Formatter` a beaucoup d'autres méthodes utiles, y compris certaines pour gérer des données structurées comme des cartes, des listes, etc., que nous n'aborderons pas ici ; consultez la documentation en ligne pour tous les détails.

Utilisation du langage de formatage dans votre propre code

Vous pouvez écrire vos propres fonctionset les macros qui acceptent les modèles de format et les arguments en utilisant la `format_args!` macro de Rustet le `std::fmt::Arguments` type. Par exemple, supposons que votre programme doive se connectermessages d'état pendant son exécution, et vous souhaitez utiliser le langage de formatage de texte de Rust pour les produire. Ce qui suit serait un début :

```
fn logging_enabled() ->bool { ... }

use std:: fs:: OpenOptions;
use std:: io::Write;

fn write_log_entry(entry: std:: fmt:: Arguments) {
    if logging_enabled() {
        // Keep things simple for now, and just
        // open the file every time.
        let mut log_file = OpenOptions::new()
            .append(true)
            .create(true)
            .open("log-file-name")
            .expect("failed to open log file");

        log_file.write_fmt(entry)
            .expect("failed to write to log");
    }
}
```

Vous pouvez appeler `write_log_entry` ainsi :

```
write_log_entry(format_args!("Hark! {:?}\n", mysterious_value));
```

Au moment de la compilation, la `format_args!` macro analyse la chaîne du modèle et la compare aux types des arguments, signalant une erreur en cas de problème. Au moment de l'exécution, il évalue les arguments et construit une `Arguments` valeur contenant toutes les informations nécessaires pour formater le texte : une forme pré-parsée du modèle, ainsi que des références partagées aux valeurs des arguments.

Construire une `Arguments` valeur n'est pas cher : c'est juste rassembler quelques pointeurs. Aucun travail de formatage n'a encore lieu, seulement la collecte des informations nécessaires pour le faire plus tard. Cela peut être important : si la journalisation n'est pas activée, tout le temps

passé à convertir des nombres en décimal, à remplir des valeurs, etc. serait perdu.

Le `File` typeimplémente le `std::io::Write` trait, dont la `write_fmt` méthode prend un `Argument` et effectue le formatage. Il écrit les résultats dans le flux sous-jacent.

Cet appel `write_log_entry` n'est pas joli. C'est là qu'une macro peut aider :

```
macro_rules! log { // no ! needed after name in macro definitions
    ($format: tt, $($arg:expr),*) => (
        write_log_entry(format_args!($format, $($arg),*))
    )
}
```

Nous couvrons les macros en détail au chapitre 21 . Pour l'instant, croyez que cela définit une nouvelle `log!` macro qui transmet ses arguments à `format_args!` puis appelle votre `write_log_entry` fonction sur la `Arguments` valeur résultante. Les macros de formatage comme `println!` , `writeln!` et `format!` sont toutes à peu près la même idée.

Vous pouvez utiliser `log!` comme ceci :

```
log!("O day and night, but this is wondrous strange! {:?}\n",
     mysterious_value);
```

Idéalement, cela semble un peu mieux.

Expressions régulières

`regex` La caisse externeest l'expression régulière officielle de Rustbibliothèque. Il fournit les fonctions habituelles de recherche et de correspondance. Il prend bien en charge Unicode, mais il peut également rechercher des chaînes d'octets. Bien qu'il ne prenne pas en charge certaines fonctionnalités que vous trouverez souvent dans d'autres packages d'expressions régulières, comme les références arrière et les modèles de recherche, ces simplifications permettent `regex` de garantir que les recherches prennent un temps linéaire dans la taille de l'expression et dans la longueur du texte. étant recherché. Ces garanties, entre autres, garantissent `regex` une utilisation sûre même avec des expressions non fiables recherchant du texte non fiable.

Dans ce livre, nous ne fournirons qu'un aperçu de `regex` ; vous devriez consulter sa documentation en ligne pour plus de détails.

Bien que la `regex` caisse ne soit pas dans `std` , elle est entretenue par l'équipe de la bibliothèque Rust, le même groupe responsable de `std` . Pour utiliser `regex` , placez la ligne suivante dans la `[dependencies]` section du fichier *Cargo.toml* de votre caisse :

```
expression régulière = "1"
```

Dans les sections suivantes, nous supposerons que vous avez mis en place ce changement.

Utilisation de base des expressions régulières

Une `Regex` valeur représente une expression régulière analyséeprêt à l'emploi. Le `Regex::new` constructeur essaie d'analyser a `&str` comme une expression régulière et renvoie a `Result` :

```
use regex::Regex;

// A semver version number, like 0.2.1.
// May contain a pre-release version suffix, like 0.2.1-alpha.
// (No build metadata suffix, for brevity.)
//
// Note use of r"..." raw string syntax, to avoid backslash blizzard.
let semver = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")?;

// Simple search, with a Boolean result.
let haystack = r#"regex = "0.2.5""#;
assert!(semver.is_match(haystack));
```

La `Regex::captures` méthoderecherche une chaîne pour la première correspondance et renvoie une `regex::Captures` valeur contenant des informations de correspondance pour chaque groupe dans l'expression :

```
// You can retrieve capture groups:
let captures = semver.captures(haystack)
    .ok_or("semver regex should have matched")?;
assert_eq!(&captures[0], "0.2.5");
assert_eq!(&captures[1], "0");
assert_eq!(&captures[2], "2");
assert_eq!(&captures[3], "5");
```

L' indexation d'une `Captures` valeur panique si le groupe demandé ne correspond pas. Pour tester si un groupe particulier correspond, vous pouvez appeler `Captures::get` , qui renvoie un `Option<regex::Match>` . Une `Match` valeur enregistre la correspondance d'un seul groupe :

```
assert_eq!(captures.get(4), None);
assert_eq!(captures.get(3).unwrap().start(), 13);
assert_eq!(captures.get(3).unwrap().end(), 14);
assert_eq!(captures.get(3).unwrap().as_str(), "5");
```

Vous pouvez parcourir toutes les correspondances d'une chaîne :

```
let haystack = "In the beginning, there was 1.0.0. \
                For a while, we used 1.0.1-beta, \
                but in the end, we settled on 1.2.4.";

let matches:Vec<&str> = semver.find_iter(haystack)
    .map(|match_| match_.as_str())
    .collect();
assert_eq!(matches, vec!["1.0.0", "1.0.1-beta", "1.2.4"]);
```

L' `find_iter` itérateurproduit une `Match` valeur pour chaque correspondance sans chevauchement de l'expression, du début à la fin de la chaîne. La `captures_iter` méthodeest similaire, mais produit des `Captures` valeurs enregistrant tous les groupes de capture. La recherche est plus lente lorsque les groupes de capture doivent être signalés, donc si vous n'en avez pas besoin, il est préférable d'utiliser l'une des méthodes qui ne les renvoie pas.

Construire des valeurs Regex paresseusement

Le `Regex::new` constructeurpeut être coûteux : la construction `Regex` d'une expression régulière de 1 200 caractères peut prendre près d'une milliseconde sur une machine de développement rapide, et même une expression triviale prend des microsecondes. Il est préférable de garder la `Regex` construction hors des boucles de calcul lourdes ; au lieu de cela, vous devez construire votre `Regex` une fois, puis réutiliser le même.

La `lazy_static` caissefournit un bon moyen de construire des valeurs statiquesparesseusement la première fois qu'ils sont utilisés. Pour commencer, notez la dépendance dans votre fichier *Cargo.toml* :

```
[dépendances]
paresseux_statique = "1"
```

Ce crate fournit une macro pour déclarer de telles variables :

```
use lazy_static::lazy_static;

lazy_static! {
    static ref SEMVER: Regex
        = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")
              .expect("error parsing regex");
}
```

La macro se développe en une déclaration d'une variable statiquenommé `SEMVER` , mais son type n'est pas exactement `Regex` . Au lieu de cela, il s'agit d'un type généré par macro qui implémente `Deref<Target=Regex>` et expose donc toutes les mêmes méthodes qu'un `Regex` . La première fois `SEMVER` est déréférencée, l'initialiseur est évalué et la valeur est enregistrée pour une utilisation ultérieure. Puisqu'il `SEMVER` s'agit d'une variable statique, et pas seulement d'une variable locale, l'initialiseur s'exécute au plus une fois par exécution du programme.

Avec cette déclaration en place, l'utilisation `SEMVER` est simple :

```
use std:: io::BufRead;

let stdin = std:: io::stdin();
for line_result in stdin.lock().lines() {
    let line = line_result?;
    if let Some(match_) = SEMVER.find(&line) {
        println!("{}", match_.as_str());
    }
}
```

Vous pouvez mettre la `lazy_static!` déclaration dans un module, ou même à l'intérieur de la fonction qui utilise le `Regex` , si c'est la portée la plus appropriée. L'expression régulière est toujours toujours compilée une seule fois par exécution du programme.

Normalisation

La plupart des utilisateurs considéreraient que le mot français pour thé, *thé* , comporte trois caractères. Cependant, Unicodea en fait deuxfaçons de représenter ce texte :

- Dans la forme *composée* , *« thé »* comprend les trois caractères `'t'` , `'h'` , et `'é'` , où `'é'` est un seul caractère Unicode avec le point de code `0xe9` .
- Dans sa forme *décomposée* , *"thé"* comprend les quatre caractères `'t'` , `'h'` , `'e'` , et `'\u{301}'` , où le `'e'` est le caractère ASCII simple, sans accent, et le point de code `0x301` est le caractère "COMBINING ACUTE ACCENT", qui ajoute un accent aigu à tout caractère qu'il suit.

Unicode ne considère ni la forme composée ni la forme décomposée de *é* comme étant la « correcte » ; il les considère plutôt comme des représentations équivalentes du même caractère abstrait. Unicode indique que les deux formulaires doivent être affichés de la même manière et que les méthodes de saisie de texte sont autorisées à produire l'un ou l'autre, de sorte que les utilisateurs ne sauront généralement pas quel formulaire ils visualisent ou tapent. (Rust vous permet d'utiliser des caractères Unicode directement dans les littéraux de chaîne, vous pouvez donc simplement écrire `"thé"` si vous ne vous souciez pas de l'encodage que vous obtenez. Ici, nous utiliserons les `\u` échappements pour plus de clarté.)

Cependant, considérés comme Rust `&str` ou `String` valeurs, `"th\u{e9}"` et `"the\u{301}"` sont complètement distincts. Ils ont des longueurs différentes, se comparent comme inégaux, ont des valeurs de hachage différentes et s'ordonnent différemment par rapport aux autres chaînes :

```
assert!("th\u{e9}" != "the\u{301}");
assert!("th\u{e9}" >  "the\u{301}");

// A Hasher is designed to accumulate the hash of a series of values,
// so hashing just one is a bit clunky.
use std:: hash:: {Hash, Hasher};
use std:: collections:: hash_map:: DefaultHasher;
fn hash<T: ?Sized + Hash>(t: &T) -> u64 {
    let mut s = DefaultHasher::new();
    t.hash(&mut s);
    s.finish()
}

// These values may change in future Rust releases.
assert_eq!(hash("th\u{e9}"),   0x53e2d0734eb1dff3);
assert_eq!(hash("the\u{301}"), 0x90d837f0a0928144);
```

De toute évidence, si vous avez l'intention de comparer du texte fourni par l'utilisateur ou de l'utiliser comme clé dans une table de hachage ou un arbre B, vous devrez d'abord mettre chaque chaîne sous une forme canonique.

Heureusement, Unicode spécifie des formes *normalisées pour les chaînes.* Chaque fois que deux chaînes doivent être traitées comme équivalentes selon les règles d'Unicode, leurs formes normalisées sont identiques caractère pour caractère. Lorsqu'ils sont encodés avec UTF-8, ils sont identiques octet par octet. Cela signifie que vous pouvez comparer des chaînes normalisées avec `==`, les utiliser comme clés dans a `HashMap` ou `HashSet`, etc., et vous obtiendrez la notion d'égalité d'Unicode.

L'absence de normalisation peut même avoir des conséquences sur la sécurité. Par exemple, si votre site Web normalise les noms d'utilisateur dans certains cas mais pas dans d'autres, vous pourriez vous retrouver avec deux utilisateurs distincts nommés `bananasflambé`, que certaines parties de votre code traitent comme le même utilisateur, mais que d'autres distinguent, ce qui entraînerait l'extension incorrecte de ses privilèges au autre. Bien sûr, il existe de nombreuses façons d'éviter ce genre de problème, mais l'histoire montre qu'il existe également de nombreuses façons de ne pas le faire.

Formulaires de normalisation

Unicode définit quatre formes normalisées, dont chacune est appropriée pour différentes utilisations. Il y a deux questions auxquelles répondre :

- Premièrement, préférez-vous que les personnages soient aussi *composés* que possible ou aussi *décomposés* que possible ?
  Par exemple, la représentation la plus composée du mot vietnamien *Phở* est la chaîne de trois caractères `"Ph\u{1edf}"`, où la marque tonale ̉et la marque de voyelle ̛sont appliquées au caractère de base "o" dans un seul caractère Unicode, `'\u{1edf}'`, que Unicode nomme consciencieusement LETTRE MINUSCULE LATINE O AVEC CORNE ET CROCHET AU-DESSUS.
  La représentation la plus décomposée divise la lettre de base et ses deux marques en trois caractères Unicode distincts : `'o'`, `'\u{31b}'` (COMBINING HORN) et `'\u{309}'` (COMBINING HOOK ABOVE), ce qui donne `"Pho\u{31b}\u{309}"`. (Chaque fois que des marques combinées apparaissent comme des caractères séparés, plutôt que comme faisant partie d'un caractère composé, toutes les formes normalisées spécifient un ordre fixe dans lequel elles doivent

apparaître, de sorte que la normalisation est bien spécifiée même lorsque les caractères ont plusieurs accents.)
La forme composée a généralement moins de problèmes de compatibilité, car elle correspond plus étroitement aux représentations que la plupart des langues utilisaient pour leur texte avant l'établissement d'Unicode. Cela peut également mieux fonctionner avec des fonctionnalités de formatage de chaîne naïves comme la `format!` macro de Rust. La forme décomposée, en revanche, peut être plus adaptée à l'affichage du texte ou à la recherche, car elle rend la structure détaillée du texte plus explicite.

- La deuxième question est la suivante : si deux séquences de caractères représentent le même texte fondamental mais diffèrent dans la manière dont le texte doit être formaté, voulez-vous les traiter comme équivalentes ou les garder distinctes ?
  Unicode a des caractères séparés pour le chiffre ordinaire `5`, le chiffre en exposant `⁵` (ou `'\u{2075}'`) et le chiffre encerclé ⑤ (ou `'\u{2464}'`), mais déclare que les trois sont *équivalents en termes de compatibilité*. De même, Unicode a un seul caractère pour la ligature *ffi* (`'\u{fb03}'`), mais déclare qu'il s'agit d'une compatibilité équivalente à la séquence de trois caractères `ffi`.
  L'équivalence de compatibilité est logique pour les recherches : une recherche de `"difficult"`, en utilisant uniquement des caractères ASCII, doit correspondre à la chaîne `"di\u{fb03}cult"`, qui utilise la ligature *ffi*. L'application de la décomposition de compatibilité à cette dernière chaîne remplacerait la ligature par les trois lettres simples `"ffi"`, ce qui faciliterait la recherche. Mais la normalisation du texte dans une forme équivalente de compatibilité peut perdre des informations essentielles, elle ne doit donc pas être appliquée avec négligence. Par exemple, il serait incorrect dans la plupart des contextes de stocker `"2⁵"` en tant que `"25"`.

La forme de normalisation Unicode C et la forme de normalisation D (NFC et NFD) utilisent les formes composées au maximum et décomposées au maximum de chaque caractère, mais n'essayez pas d'unifier les séquences équivalentes de compatibilité. Les formes de normalisation NFKC et NFKD sont comme NFC et NFD, mais normalisent toutes les séquences équivalentes de compatibilité à un simple représentant de leur classe.

Le « Modèle de personnage pour le World Wide Web » du World Wide Web Consortium recommande d'utiliser NFC pour tout le contenu. L'annexe Unicode Identifier and Pattern Syntax suggère d'utiliser NFKC pour les identifiants dans les langages de programmation et propose des principes pour adapter la forme si nécessaire.

La caisse de normalisation unicode

`unicode-normalization` La caisse de Rustfournit un trait qui ajoute des méthodes pour `&str` mettre le texte dans l'une des quatre formes normalisées. Pour l'utiliser, ajoutez la ligne suivante à la `[dependencies]` section de votre fichier *Cargo.toml* :

```
normalisation unicode = "0.1.17"
```

Avec cette déclaration en place, a `&str` a quatre nouvelles méthodes qui renvoient des itérateurs sur une forme normalisée particulière de la chaîne :

```
use unicode_normalization::UnicodeNormalization;

// No matter what representation the left-hand string uses
// (you shouldn't be able to tell just by looking),
// these assertions will hold.
assert_eq!("Phở".nfd().collect:: <String>(), "Pho\u{31b}\u{309}");
assert_eq!("Phở".nfc().collect::<String>(), "Ph\u{1edf}");

// The left-hand side here uses the "ffi" ligature character.
assert_eq!("① Di\u{fb03}culty".nfkc().collect::<String>(), "1 Difficulty"
```

Prendre une chaîne normalisée et la normaliser à nouveau sous la même forme est garanti pour renvoyer un texte identique.

Bien que toute sous-chaîne d'une chaîne normalisée soit elle-même normalisée, la concaténation de deux chaînes normalisées n'est pas nécessairement normalisée : par exemple, la deuxième chaîne peut commencer par des caractères de combinaison qui doivent être placés avant les caractères de combinaison à la fin de la première chaîne.

Tant qu'un texte n'utilise aucun point de code non attribué lorsqu'il est normalisé, Unicode promet que sa forme normalisée ne changera pas dans les futures versions de la norme. Cela signifie que les formulaires normalisés peuvent généralement être utilisés en toute sécurité dans un stockage persistant, même si la norme Unicodeévolue.