Chapitre 15. Itérateurs

> *C'était la fin d'une très longue journée.*
>
> —Phil

Un *itérateur*est une valeur qui produit une séquence de valeurs, généralement sur laquelle une boucle doit fonctionner. La bibliothèque standard de Rust fournit des itérateurs qui traversent des vecteurs, des chaînes, des tables de hachage et d'autres collections, mais aussi des itérateurs pour produire des lignes de texte à partir d'un flux d'entrée, des connexions arrivant sur un serveur réseau, des valeurs reçues d'autres threads sur un canal de communication, etc. sur. Et bien sûr, vous pouvez implémenter des itérateurs à vos propres fins. La boucle de Rust `for` fournit une syntaxe naturelle pour l'utilisation des itérateurs, mais les itérateurs eux-mêmes fournissent également un riche ensemble de méthodes pour mapper, filtrer, joindre, collecter, etc.

Les itérateurs de Rust sont flexibles, expressifs et efficaces. Considérez la fonction suivante, qui renvoie la somme des premiers `n` entiers positifs (souvent appelé le *`n` ème nombre de triangle*):

```
fn triangle(n: i32) ->i32 {
    let mut sum = 0;
    for i in 1..=n {
        sum += i;
    }
    sum
}
```

L'expression `1..=n` est une `RangeInclusive<i32>` valeur. A `RangeInclusive<i32>` est un itérateur qui produit les entiers de sa valeur de départ à sa valeur de fin (toutes deux incluses), vous pouvez donc l'utiliser comme opérande de la `for` boucle pour additionner les valeurs de `1` à `n`.

Mais les itérateurs ont aussi une `fold` méthode, que vous pouvez utiliser dans la définition équivalente :

```
fn triangle(n: i32) ->i32 {
    (1..=n).fold(0, |sum, item| sum + item)
}
```

Starting with `0` as the running total, `fold` takes each value that `1..=n` produces and applies the closure `|sum, item| sum + item` to the running total and the value. The closure's return value is taken as the new running total. The last value it returns is what `fold` itself returns—in this case, the total of the entire sequence. This may look strange if you're used to `for` and `while` loops, but once you've gotten used to it, `fold` is a legible and concise alternative.

C'est un tarif assez standard pour les langages de programmation fonctionnels, qui privilégient l'expressivité. Mais les itérateurs de Rust ont été soigneusement conçus pour garantir que le compilateur puisse également les traduire en un excellent code machine. Dans une version de version de la deuxième définition présentée précédemment, Rust connaît la définition de `fold` et l'intègre dans `triangle`. Ensuite, la fermeture `|sum, item| sum + item` est intégrée à cela. Enfin, Rust examine le code combiné et reconnaît qu'il existe un moyen plus simple d'additionner les nombres de un à `n` : la somme est toujours égale à `n * (n+1) / 2`. Rust traduit le corps entier de `triangle`, boucle, fermeture, et tout, en une seule instruction de multiplication et quelques autres bits d'arithmétique.

Il se trouve que cet exemple implique une arithmétique simple, mais les itérateurs fonctionnent également bien lorsqu'ils sont utilisés de manière intensive. Ils sont un autre exemple de Rust fournissant des abstractions flexibles qui imposent peu ou pas de surcharge lors d'une utilisation typique.

Dans ce chapitre, nous expliquerons :

- Les traits `Iterator` et `IntoIterator`, qui sont à la base des itérateurs de Rust
- Les trois étapes d'un pipeline d'itérateur typique: création d'un itérateur à partir d'une sorte de source de valeur ; adapter une sorte d'itérateur à une autre en sélectionnant ou en traitant des valeurs au fur et à mesure de leur passage ; puis en consommant les valeurs produites par l'itérateur
- Comment implémenter des itérateurs pour vos propres types

Il existe de nombreuses méthodes, vous pouvez donc parcourir une section une fois que vous avez compris l'idée générale. Mais les itérateurs sont très courants dans Rust idiomatique, et il est essentiel de se familiariser avec les outils qui les accompagnent pour maîtriser le langage.

Les traits Iterator et IntoIterator

Un itérateurest une valeur qui implémente le `std::iter::Iterator` trait :

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ... // many default methods
}
```

`Item` est le type de valeur produit par l'itérateur. La `next` méthode renvoie soit `Some(v)` , où `v` est la valeur suivante de l'itérateur, soit renvoie `None` pour indiquer la fin de la séquence. Ici, nous avons omis `Iterator` les nombreuses méthodes par défaut de; nous les aborderons individuellement dans le reste de ce chapitre.

S'il existe un moyen naturel d'itérer sur un type, ce type peut implémenter `std::iter::IntoIterator` , dont la `into_iter` méthode prend une valeur et renvoie un itérateur dessus :

```
trait IntoIterator where Self:: IntoIter: Iterator<Item=Self:: Item> {
    type Item;
    type IntoIter: Iterator;
    fn into_iter(self) -> Self::IntoIter;
}
```

`IntoIter` est le genrede la valeur de l'itérateur lui-même, et `Item` est le type de valeur qu'il produit. Nous appelons tout type qui implémente `IntoIterator` un *itérable*, car c'est quelque chose que vous pouvez répéter si vous le demandez.

La boucle de Rust `for` rassemble bien toutes ces parties. Pour itérer sur les éléments d'un vecteur, vous pouvez écrire :

```
println!("There's:");
let v = vec!["antimony", "arsenic", "aluminum", "selenium"];

for element in &v {
    println!("{}", element);
}
```

Sous le capot, chaque `for` boucle n'est qu'un raccourci pour les appels `IntoIterator` et `Iterator` les méthodes :

```
let mut iterator = (&v).into_iter();
while let Some(element) = iterator.next() {
    println!("{}", element);
}
```

La `for` boucle utilise `IntoIterator::into_iter` pour convertir son opérande `&v` en un itérateur , puis appelle `Iterator::next` à plusieurs reprises. Chaque fois que renvoie `Some(element)` , la `for` boucle exécute son corps ; et s'il retourne `None` , la boucle se termine.

En gardant cet exemple à l'esprit, voici une terminologie pour les itérateurs :

- Comme nous l'avons dit, un *itérateur* est tout type qui implémente `Iterator` .
- Un *itérable* est tout type qui implémente `IntoIterator` : vous pouvez obtenir un itérateur dessus en appelant sa `into_iter` méthode. La référence vectorielle `&v` est l'itérable dans ce cas.
- Un itérateur *produit des* valeurs.
- Les valeurs produites par un itérateur sont des *items* . Ici, les éléments sont `"antimony"` , `"arsenic"` , etc.
- Le code qui reçoit les éléments produits par un itérateur est le *consommateur* . Dans cet exemple, la `for` boucle est le consommateur.

Bien qu'une `for` boucleappelle toujours `into_iter` son opérande, vous pouvez également passer `for` directement les itérateurs aux boucles ; cela se produit lorsque vous bouclez sur un `Range` , par exemple. Tous les itérateurs implémentent automatiquement `IntoIterator` , avec une `into_iter` méthode qui renvoie simplement l'itérateur.

Si vous appelez à nouveau la méthode d'un itérateur `next` après son retour `None` , le `Iterator` trait ne spécifie pas ce qu'il doit faire. La plupart des itérateurs reviendront à `None` nouveau, mais pas tous. (Si cela cause des problèmes, l' `fuse` adaptateur couvert de "fusible" peut aider.)

Création d'itérateurs

La rouilleLa documentation de la bibliothèque standard explique en détail le type d'itérateurs que chaque type fournit, mais la bibliothèque suit certaines conventions générales pour vous aider à vous orienter et à trouver ce dont vous avez besoin.

Méthodes iter et iter_mut

La plupart des collectestypes fournissent `iter` et `iter_mut` méthodesqui renvoient les itérateurs naturels sur le type, produisant une référence partagée ou modifiable à chaque élément. Les tranches de tableau aiment `&[T]` et `&mut [T]` ont `iter` et `iter_mut` les méthodes aussi. Ces méthodes sont le moyen le plus courant d'obtenir un itérateur, si vous ne laissez pas une `for` boucle s'en occuper pour vous :

```
let v = vec![4, 20, 12, 8, 6];
let mut iterator = v.iter();
assert_eq!(iterator.next(), Some(&4));
assert_eq!(iterator.next(), Some(&20));
assert_eq!(iterator.next(), Some(&12));
assert_eq!(iterator.next(), Some(&8));
assert_eq!(iterator.next(), Some(&6));
assert_eq!(iterator.next(), None);
```

Le type d'élément de cet itérateur est `&i32` : chaque appel à `next` produit une référence à l'élément suivant, jusqu'à ce que nous atteignions la fin du vecteur.

Chaque type est libre de mise en œuvre `iter` et `iter_mut` de la manière la plus logique pour son objectif. La `iter` méthode sur les `std::path::Path` retoursun itérateur qui produit un composant de chemin à la fois :

```
use std:: ffi:: OsStr;
use std:: path::Path;

let path = Path:: new("C:/Users/JimB/Downloads/Fedora.iso");
let mut iterator = path.iter();
assert_eq!(iterator.next(), Some(OsStr:: new("C:")));
assert_eq!(iterator.next(), Some(OsStr:: new("Users")));
assert_eq!(iterator.next(), Some(OsStr::new("JimB")));
...
```

Le type d'élément de cet itérateur est `&std::ffi::OsStr` , une tranche empruntée d'une chaîne du type accepté par les appels du système d'exploitation.

S'il existe plusieurs façons courantes d'itérer sur un type, le type fournit généralement des méthodes spécifiques pour chaque type de parcours, car une `iter` méthode simple serait ambiguë. Par exemple, il n'y a pas `iter` de méthode sur le `&str` type tranche de chaîne. Au lieu de cela, if `s` est un `&str` , puis `s.bytes()` renvoie un itérateur qui produit chaque

octet de `s`, alors `s.chars()` qu'il interprète le contenu comme UTF-8 et produit chaque caractère Unicode.

Implémentations IntoIterator

Lorsqu'un genreimplements `IntoIterator`, vous pouvez appeler sa `into_iter` méthode vous-même, comme le `for` ferait une boucle :

```
// You should usually use HashSet, but its iteration order is
// nondeterministic, so BTreeSet works better in examples.
use std:: collections:: BTreeSet;
let mut favorites = BTreeSet::new();
favorites.insert("Lucy in the Sky With Diamonds".to_string());
favorites.insert("Liebesträume No. 3".to_string());

let mut it = favorites.into_iter();
assert_eq!(it.next(), Some("Liebesträume No. 3".to_string()));
assert_eq!(it.next(), Some("Lucy in the Sky With Diamonds".to_string()));
assert_eq!(it.next(), None);
```

La plupart des collections fournissent en fait plusieurs implémentations de `IntoIterator`, pour les références partagées(`&T`), les références mutables (`&mut T`) et les déplacements (`T`) :

- Étant donné une *référence partagée*à la collection, `into_iter` renvoie un itérateur qui produit des références partagées à ses éléments. Par exemple, dans le code précédent, `(&favorites).into_iter()` renverrait un itérateur dont le `Item` type est `&String`.
- Étant donné une *référence mutable*à la collection, `into_iter` renvoie un itérateur qui produit des références mutables aux éléments. Par exemple, si `vector` est certains `Vec<String>`, l'appel `(&mut vector).into_iter()` renvoie un itérateur dont le `Item` type est `&mut String`.
- Lors du passage de la collection *par valeur*, `into_iter` renvoie un itérateur qui s'approprie la collection et renvoie les éléments par valeur ; la propriété des articles passe de la collection au consommateur, et la collection d'origine est consommée dans le processus. Par exemple, l'appel `favorites.into_iter()` dans le code précédent renvoie un itérateur qui produit chaque chaîne par valeur ; le consommateur reçoit la propriété de chaque chaîne. Lorsque l'itérateur est supprimé, tous les éléments restants dans le `BTreeSet` sont également supprimés et l'enveloppe désormais vide de l'ensemble est supprimée.

Depuis une `for` boucles'applique `IntoIterator::into_iter` à son opérande, ces trois implémentations créent les idiomes suivants pour itérer sur des références partagées ou modifiables à une collection, ou consommer la collection et s'approprier ses éléments :

```
for element in &collection { ... }
for element in &mut collection { ... }
for element in collection { ... }
```

Chacun de ces résultats entraîne simplement un appel à l'une des `IntoIterator` implémentations répertoriées ici.

Tous les types ne fournissent pas les trois implémentations. Par exemple, `HashSet`, `BTreeSet` et `BinaryHeap` ne s'implémentent pas `IntoIterator` sur des références mutables, car la modification de leurs éléments violerait probablement les invariants du type : la valeur modifiée pourrait avoir une valeur de hachage différente, ou être ordonnée différemment par rapport à ses voisins, donc la modifier laisserait il est mal placé. D'autres types prennent en charge la mutation, mais seulement partiellement. Par exemple, `HashMap` et `BTreeMap` produisent une référence mutable aux valeurs de leurs entrées, mais uniquement des références partagées à leurs clés, pour des raisons similaires à celles données précédemment.

Le principe général est que l'itération doit être efficace et prévisible, donc plutôt que de fournir des implémentations qui sont coûteuses ou qui pourraient présenter un comportement surprenant (par exemple, ressasser les `HashSet` entrées modifiées et éventuellement les rencontrer à nouveau plus tard dans l'itération), Rust les omet entièrement.

Tranchesmettre en œuvre deux des trois `IntoIterator` variantes ; puisqu'ils ne sont pas propriétaires de leurs éléments, il n'y a pas de cas « par valeur ». Au lieu de cela, `into_iter` for `&[T]` et `&mut [T]` renvoie un itérateur qui produit des références partagées et modifiables aux éléments. Si vous imaginez le type de tranche sous-jacent `[T]` comme une collection quelconque, cela s'intègre parfaitement dans le modèle global.

Vous avez peut-être remarqué que les deux premières `IntoIterator` variantes, pour les références partagées et mutables, sont équivalentes à appeler `iter` ou `iter_mut` sur le référent. Pourquoi Rust fournit-il les deux ?

`IntoIterator` c'est ce qui fait les `for` bouclestravail, donc c'est évidemment nécessaire. Mais lorsque vous n'utilisez pas de `for` boucle, il est

plus clair d'écrire `favorites.iter()` que `(&favorites).into_iter()`. L'itération par référence partagée est quelque chose dont vous aurez fréquemment besoin, donc `iter` et `iter_mut` sont toujours précieux pour leur ergonomie.

`IntoIterator` peut également être utile en génériquecode : vous pouvez utiliser une limite comme `T: IntoIterator` pour restreindre la variable de type `T` aux types qui peuvent être itérés. Ou, vous pouvez écrire `T: IntoIterator<Item=U>` pour exiger davantage que l'itération produise un type particulier `U`. Par exemple, cette fonction vide les valeurs de tout itérable dont les éléments sont imprimables au `"{:?}"` format :

```
use std:: fmt::Debug;

fn dump<T, U>(t: T)
    where T: IntoIterator<Item=U>,
          U:Debug
{
    for u in t {
        println!("{:?}", u);
    }
}
```

Vous ne pouvez pas écrire cette fonction générique en utilisant `iter` and `iter_mut`, car ce ne sont pas des méthodes d'aucun trait : la plupart des types itérables ont simplement des méthodes portant ces noms.

from_fn et successeurs

Un simpleet la manière générale de produire une séquence de valeurs est de fournir une fermeture qui les renvoie.

Étant donné une fonction renvoyant `Option<T>`, `std::iter::from_fn` renvoie un itérateur qui appelle simplement la fonction pour produire ses éléments. Par exemple:

```
use rand:: random; // In Cargo.toml dependencies: rand = "0.7"
use std:: iter::from_fn;

// Generate the lengths of 1000 random line segments whose endpoints
// are uniformly distributed across the interval [0, 1]. (This isn't a
// distribution you're going to find in the `rand_distr` crate, but
// it's easy to make yourself.)
let lengths: Vec<f64> =
    from_fn(|| Some((random:: <f64>() - random::<f64>()).abs()))
```

```
        .take(1000)
        .collect();
```

Cela appelle `from_fn` à faire un itérateur produisant des nombres aléatoires. Puisque l'itérateur retourne toujours `Some`, la séquence ne se termine jamais, mais nous l'appelons `take(1000)` pour la limiter aux 1 000 premiers éléments. Construit ensuite `collect` le vecteur à partir de l'itération résultante. C'est un moyen efficace de construire des vecteurs initialisés ; nous expliquons pourquoi dans « Construire des collections : collect et FromIterator », plus loin dans ce chapitre.

Si chaque élément dépend du précédent, la `std::iter::successors` fonction fonctionne bien. Vous fournissez un élément initial et une fonction qui prend un élément et renvoie un élément `Option` suivant. S'il renvoie `None`, l'itération se termine. Par exemple, voici une autre façon d'écrire la `escape_time` fonctionde notre set plotter de Mandelbrot au chapitre 2 :

```
use num:: Complex;
use std:: iter::successors;

fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let zero = Complex { re: 0.0, im:0.0 };
    successors(Some(zero), |&z| { Some(z * z + c) })
        .take(limit)
        .enumerate()
        .find(|(_i, z)| z.norm_sqr() > 4.0)
        .map(|(i, _z)| i)
}
```

En commençant par zéro, l' `successors` appel produit une séquence de points sur le plan complexe en élevant à plusieurs reprises le dernier point au carré et en ajoutant le paramètre `c`. Lors du traçage de l'ensemble de Mandelbrot, nous voulons voir si cette séquence orbite près de l'origine pour toujours ou s'envole vers l'infini. L'appel `take(limit)` établit une limite sur la durée pendant laquelle nous poursuivrons la séquence et `enumerate` numérote chaque point, transformant chaque point `z` en un tuple `(i, z)`. Nous `find` cherchons le premier point qui s'éloigne suffisamment de l'origine pour s'échapper. La `find` méthode renvoie un `Option` : `Some((i, z))` s'il existe, ou `None` sinon. L'appel à `Option::map` se transforme `Some((i, z))` en `Some(i)`, mais retourne `None` inchangé : c'est exactement la valeur de retour que nous voulons.

Les deux `from_fn` et `successors` acceptent les `FnMut` fermetures, afin que vos fermetures puissent capturer et modifier les variables des éten-

dues environnantes. Par exemple, cette `fibonacci` fonction utilise une `move` fermeture pour capturer une variable et l'utiliser comme état d'exécution :

```
fn fibonacci() -> impl Iterator<Item=usize> {
    let mut state = (0, 1);
    std:: iter::from_fn(move || {
        state = (state.1, state.0 + state.1);
        Some(state.0)
    })
}

assert_eq!(fibonacci().take(8).collect::<Vec<_>>(),
           vec![1, 1, 2, 3, 5, 8, 13, 21]);
```

Une note de prudence : les méthodes `from_fn` et `successors` sont suffisamment flexibles pour que vous puissiez transformer à peu près n'importe quelle utilisation d'itérateurs en un seul appel à l'un ou à l'autre, en passant des fermetures complexes pour obtenir le comportement dont vous avez besoin. Mais cela néglige l'opportunité qu'offrent les itérateurs de clarifier la façon dont les données circulent dans le calcul et d'utiliser des noms standard pour les modèles communs. Assurez-vous de vous être familiarisé avec les autres méthodes d'itération de ce chapitre avant de vous appuyer sur ces deux ; il y a souvent des façons plus agréables de faire le travail.

Méthodes de vidange

De nombreux types de collections fournissent une `drain` méthodequi prend une référence mutable à la collection et renvoie un itérateur qui transmet la propriété de chaque élément au consommateur. Cependant, contrairement à la `into_iter()` méthode, qui prend la collection par valeur et la consomme, `drain` emprunte simplement une référence mutable à la collection, et lorsque l'itérateur est supprimé, il supprime tous les éléments restants de la collection et la laisse vide.

Sur les types qui peuvent être indexés par une plage, comme `String` s, vectors et `VecDeque` s, la `drain` méthode prend une plage d'éléments à supprimer, plutôt que de vider toute la séquence :

```
let mut outer = "Earth".to_string();
let inner = String::from_iter(outer.drain(1..4));

assert_eq!(outer, "Eh");
assert_eq!(inner, "art");
```

Si vous avez besoin de vider toute la séquence, utilisez la plage complète, `..`, comme argument.

Autres sources d'itérateurs

Les sections précédentes concernent principalement les types de collection comme les vecteurs et `HashMap`, mais il existe de nombreux autres types dans la normebibliothèque prenant en charge l'itération. Le tableau 15-1 résume les plus intéressantes, mais il y en a bien d'autres. Nous couvrons certaines de ces méthodes plus en détail dans les chapitres consacrés aux types spécifiques(à savoir, les chapitres 16 , 17 et 18).

Tableau 15-1. Autres itérateurs de la bibliothèque standard

Type ou caractère	Expression	Remarques
`std::ops::Range`	`1..10`	Les points de terminaison doivent être de type entier pour être itérables. La plage inclut la valeur de début et exclut la valeur de fin.
	`(1..10).step_by(2)`	Produit `1, 3, 5, 7, 9.`
`std::ops::RangeFrom`	`1..`	Itération illimitée. Début doit être un entier. Peut paniquer ou déborder si la valeur atteint la limite du type.
`std::ops::RangeInclusive`	`1..=10`	Comme `Range`, mais inclut la valeur finale.
`Option<T>`	`Some(10).iter()`	Se comporte comme un vecteur dont la longueur est soit 0 (`None`) soit 1 (`Some(v)`).
`Result<T, E>`	`Ok("blah").iter()`	Similaire à `Option`, produisant des `Ok` valeurs.
`Vec<T>, &[T]`	`v.windows(16)`	Produit chaque tranche contiguë de la longueur donnée, de gauche à droite. Les fenêtres se chevauchent.
	`v.chunks(16)`	Produit des tranches contiguës sans chevauchement de la longueur donnée, de gauche à droite.
	`v.chunks_mut(1024)`	Comme `chunks`, mais les tranches sont modifiables.

Type ou caractère	Expression	Remarques
	`v.split(|byte| byte & 1 != 0)`	Produit des tranches séparées par des éléments qui correspondent au prédicat donné.
	`v.split_mut(...)`	Comme ci-dessus, mais produit des tranches modifiables.
	`v.rsplit(...)`	Comme `split`, mais produit des tranches de droite à gauche.
	`v.splitn(n, ...)`	Comme `split`, mais produit au plus des `n` tranches.
`String`, `&str`	`s.bytes()`	Produit les octets du formulaire UTF-8.
	`s.chars()`	Produit le `char`s représenté par UTF-8.
	`s.split_whitespace()`	Divise la chaîne par des espaces blancs et produit des tranches de caractères sans espace.
	`s.lines()`	Produit des tranches des lignes de la chaîne.
	`s.split('/')`	Divise la chaîne sur un modèle donné, produisant les tranches entre les correspondances. Les modèles peuvent être de plusieurs choses : caractères, chaînes, fermetures.
	`s.matches(char::is_numeric)`	Produit des tranches correspondant au motif donné.

Type ou caractère	Expression	Remarques
`std::collections::HashMap`, `std::collections::BTreeMap`	`map.keys()`, `map.values()`	Produit des références partagées aux clés ou aux valeurs de la carte.
	`map.values_mut()`	Produit des références mutables aux valeurs des entrées.
`std::collections::HashSet`, `std::collections::BTreeSet`	`set1.union(set2)`	Produit des références partagées aux éléments d'union de `set1` et `set2`.
	`set1.intersection(set2)`	Produit des références partagées aux éléments d'intersection de `set1` et `set2`.
`std::sync::mpsc::Receiver`	`recv.iter()`	Produit des valeurs envoyées depuis un autre thread sur le correspondant `Sender`.
`std::io::Read`	`stream.bytes()`	Produit des octets à partir d'un flux d'E/S.
	`stream.chars()`	Analyse le flux au format UTF-8 et produit `char`s.
`std::io::BufRead`	`bufstream.lines()`	Analyse le flux en UTF-8, produit des lignes en `String`s.
	`bufstream.split(0)`	Divise le flux sur un octet donné, produit des `Vec<u8>` tampons inter-octets.

Type ou caractère	Expression	Remarques
`std::fs::ReadDir`	`std::fs::read_dir(path)`	Produit des entrées de répertoire.
`std::net::TcpListener`	`listener.incoming()`	Produit des connexions réseau entrantes.
Fonctions gratuites	`std::iter::empty()`	Retourne `None` immédiatement.
	`std::iter::once(5)`	Produit la valeur donnée puis se termine.
	`std::iter::repeat("#9")`	Produit la valeur donnée pour toujours.

Adaptateurs d'itérateur

Une fois que vous avez un itérateur en main, le `Iterator` trait fournit une large sélection de *méthodes d'adaptation,* ou simplement des *adaptateurs* , qui consomment un itérateur et en construisent un nouveau avec des comportements utiles. Pour voir comment fonctionnent les adaptateurs, nous commencerons par deux des adaptateurs les plus populaires, `map` et `filter` . Ensuite, nous couvrirons le reste de la boîte à outils de l'adaptateur, couvrant presque toutes les façons imaginables de créer des séquences de valeurs à partir d'autres séquences : troncature, saut, combinaison, inversion, concaténation, répétition, etc.

carte et filtre

L' adaptateur `Iterator` du trait `map` vous permet de transformer un itérateur en appliquant une fermeture à ses éléments. L' `filter` adaptateurvous permet de filtrer les éléments d'un itérateur, en utilisant une fermeture pour décider lesquels conserver et lesquels supprimer.

Par exemple, supposons que vous itérez sur des lignes de texte et que vous souhaitiez omettre les espaces de début et de fin de chaque ligne. La `str::trim` méthode de la bibliothèque standard supprime les espaces blancs de début et de fin d'un seul `&str` , renvoyant un nouveau, coupé, `&str` qui emprunte à l'original. Vous pouvez utiliser l' `map` adaptateur pour appliquer `str::trim` à chaque ligne de l'itérateur :

```
let text = "  ponies  \n   giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .collect();
assert_eq!(v, ["ponies", "giraffes", "iguanas", "squid"]);
```

L' `text.lines()` appel renvoie un itérateur qui produit les lignes de la chaîne. L'appel `map` à cet itérateur renvoie un deuxième itérateur qui s'applique `str::trim` à chaque ligne et produit les résultats sous forme d'éléments. Enfin, `collect` rassemble ces éléments dans un vecteur.

L'itérateur qui `map` revient est, bien sûr, lui-même un candidat pour une adaptation ultérieure. Si vous souhaitez exclure les iguanes du résultat, vous pouvez écrire ce qui suit :

```
let text = "  ponies  \n   giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .filter(|s| *s != "iguanas")
    .collect();
assert_eq!(v, ["ponies", "giraffes", "squid"]);
```

Ici, `filter` renvoie un troisième itérateur qui produit uniquement les éléments de l' `map` itérateur pour lesquels la fermeture `|s| *s != "iguanas"` renvoie `true` . Une chaîne d'adaptateurs d'itérateurs est comme un pipeline dans le shell Unix : chaque adaptateur a un seul objectif, et il est clair comment la séquence est transformée lorsque l'on lit de gauche à droite.

Les signatures de ces adaptateurs sont les suivantes :

```
fn map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self:: Item) ->B;

fn filter<P>(self, predicate: P) -> impl Iterator<Item=Self:: Item>
    where Self: Sized, P: FnMut(&Self:: Item) ->bool;
```

Dans la bibliothèque standard, `map` et `filter` renvoient en fait des types opaques spécifiques `struct` nommés `std::iter::Map` et `std::iter::Filter`. Cependant, le simple fait de voir leurs noms n'est pas très informatif, donc dans ce livre, nous allons simplement écrire à la `-> impl Iterator<Item=...>` place, car cela nous dit ce que nous voulons vraiment savoir : la méthode renvoie un `Iterator` qui produit des éléments du type donné.

Étant donné que la plupart des adaptateurs prennent `self` par valeur, ils `Self` doivent être `Sized` (ce que sont tous les itérateurs courants).

Un `map` itérateur transmet chaque élément à sa fermeture par valeur et, à son tour, transmet la propriété du résultat de la fermeture à son consommateur. Un `filter` itérateur passe chaque élément à sa fermeture par référence partagée, conservant la propriété au cas où l'élément serait sélectionné pour être transmis à son consommateur. C'est pourquoi l'exemple doit déréférencer `s` pour le comparer avec `"iguanas"` : le `filter` type de l'élément de l'itérateur est `&str`, donc le type de l'argument de la fermeture `s` est `&&str`.

Il y a deux points importants à noter à propos des adaptateurs d'itérateur.

Tout d'abord, le simple fait d'appeler un adaptateur sur un itérateur ne consomme aucun élément ; il renvoie simplement un nouvel itérateur, prêt à produire ses propres éléments en puisant dans le premier itérateur si nécessaire. Dans une chaîne d'adaptateurs, la seule façon d'effectuer un travail est de faire appel `next` à l'itérateur final.

Ainsi, dans notre exemple précédent, l'appel de méthode `text.lines()` lui-même n'analyse aucune ligne de la chaîne ; il renvoie simplement un itérateur qui *analyserait* les lignes si demandé. De même, `map` et `filter` renvoyez simplement de nouveaux itérateurs qui mapperaient *ou* filtreraient si demandé. Aucun travail n'a lieu tant que l' itérateur n'a pas `collect` commencé à être appelé . `next filter`

Ce point est particulièrement important si vous utilisez des adaptateurs qui ont des effets secondaires. Par exemple, ce code n'imprime rien du tout :

```
["earth", "water", "air", "fire"]
    .iter().map(|elt| println!("{}", elt));
```

L' `iter` appel renvoie un itérateur sur les éléments du tableau et l' `map` appel renvoie un deuxième itérateur qui applique la fermeture à

chaque valeur produite par le premier. Mais il n'y a rien ici qui exige réellement une valeur de toute la chaîne, donc aucune `next` méthode ne fonctionne jamais. En fait, Rust vous avertira à ce sujet :

```
warning: unused `std::iter::Map` that must be used
  |
7 | /     ["earth", "water", "air", "fire"]
8 | |         .iter().map(|elt| println!("{}", elt));
  | |_____________________________________________^
  |
  = note: iterators are lazy and do nothing unless consumed
```

Le terme « paresseux » dans le message d'erreur n'est pas un terme désobligeant ; c'est juste du jargon pour tout mécanisme qui retarde un calcul jusqu'à ce que sa valeur soit nécessaire. C'est la convention de Rust que les itérateurs doivent faire le travail minimum nécessaire pour satisfaire chaque appel à `next` ; dans l'exemple, il n'y a aucun appel de ce type, donc aucun travail n'a lieu.

Le deuxième point important est que les adaptateurs d'itérateur sont une abstraction sans surcharge. Étant donné que `map`, `filter` et leurs compagnons sont génériques, leur application à un itérateur spécialise leur code pour le type d'itérateur spécifique impliqué. Cela signifie que Rust dispose de suffisamment d'informations pour intégrer la méthode de chaque itérateur `next` dans son consommateur, puis traduire l'intégralité de l'arrangement en code machine en tant qu'unité. Ainsi, la chaîne `lines` // `map` d' `filter` itérateurs que nous avons montrée précédemment est aussi efficace que le code que vous écririez probablement à la main :

```
for line in text.lines() {
    let line = line.trim();
    if line != "iguanas" {
        v.push(line);
    }
}
```

Le reste de cette section couvre les différents adaptateurs disponibles sur le `Iterator` trait.

filter_map et flat_map

L' `map` adaptateurconvient dans les situations où chaque élément entrant produit un élément sortant. Mais que se passe-t-il si vous souhaitez sup-

primer certains éléments de l'itération au lieu de les traiter ou remplacer des éléments uniques par zéro ou plusieurs éléments ? Les adaptateurs `filter_map` et `flat_map` vous accorder cette flexibilité.

L' `filter_map` adaptateur est similaire à `map` sauf qu'il permet à sa fermeture de transformer l'élément en un nouvel élément (comme le `map` fait) ou de supprimer l'élément de l'itération. Ainsi, c'est un peu comme une combinaison de `filter` et `map` . Sa signature est la suivante :

```
fn filter_map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self:: Item) ->Option<B>;
```

C'est la même chose que `map` la signature de , sauf qu'ici la fermeture renvoie `Option<B>` , pas simplement `B` . Lorsque la fermeture renvoie `None` , l'élément est supprimé de l'itération ; lorsqu'il retourne `Some(b)` , alors `b` est l'élément suivant produit par l' `filter_map` itérateur.

Par exemple, supposons que vous souhaitiez rechercher dans une chaîne des mots séparés par des espaces pouvant être analysés comme des nombres, et traiter les nombres en supprimant les autres mots. Tu peux écrire:

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

let text = "1\nfrond .25  289\n3.1415 estuary\n";
for number in text
    .split_whitespace()
    .filter_map(|w| f64::from_str(w).ok())
{
    println!("{:4.2}", number.sqrt());
}
```

Cela imprime ce qui suit :

```
1.00
0.50
17.00
1.77
```

La fermeture donnée à `filter_map` essaie d'analyser chaque tranche séparée par des espaces en utilisant `f64::from_str` . Cela renvoie un `Result<f64, ParseFloatError>` , qui `.ok()` se transforme en un `Option<f64>` : une erreur d'analyse devient `None` , tandis qu'un résultat

d'analyse réussi devient `Some(v)`. L' `filter_map` itérateur supprime toutes les `None` valeurs et produit la valeur `v` pour chacune `Some(v)`.

Mais quel est l'intérêt de fusionner `map` en `filter` une seule opération comme celle-ci, au lieu d'utiliser directement ces adaptateurs ? L' `filter_map` adaptateur montre sa valeur dans des situations comme celle qui vient d'être montrée, lorsque la meilleure façon de décider d'inclure ou non l'élément dans l'itération est d'essayer réellement de le traiter. Vous pouvez faire la même chose avec seulement `filter` et `map`, mais c'est un peu disgracieux :

```
text.split_whitespace()
    .map(|w| f64::from_str(w))
    .filter(|r| r.is_ok())
    .map(|r| r.unwrap())
```

Vous pouvez considérer l' `flat_map` adaptateur comme continuant dans la même veine que `map` et `filter_map`, sauf que maintenant la fermeture peut renvoyer non seulement un élément (comme avec `map`) ou zéro ou un élément (comme avec `filter_map`), mais une séquence de n'importe quel nombre d'éléments. L' `flat_map` itérateur produit la concaténation des séquences renvoyées par la fermeture.

La signature de `flat_map` est montrée ici :

```
fn flat_map<U, F>(self, f: F) -> impl Iterator<Item=U:: Item>
    where F: FnMut(Self:: Item) -> U, U:IntoIterator;
```

La fermeture passée à `flat_map` doit renvoyer un itérable, mais n'importe quel type d'itérable fera l'affaire. [1]

Par exemple, supposons que nous ayons une table mappant les pays à leurs principales villes. Étant donné une liste de pays, comment pouvons-nous itérer sur leurs principales villes ?

```
use std:: collections::HashMap;

let mut major_cities = HashMap::new();
major_cities.insert("Japan", vec!["Tokyo", "Kyoto"]);
major_cities.insert("The United States", vec!["Portland", "Nashville"]);
major_cities.insert("Brazil", vec!["São Paulo", "Brasília"]);
major_cities.insert("Kenya", vec!["Nairobi", "Mombasa"]);
major_cities.insert("The Netherlands", vec!["Amsterdam", "Utrecht"]);

let countries = ["Japan", "Brazil", "Kenya"];
```

```
for &city in countries.iter().flat_map(|country| &major_cities[country]) {
    println!("{}", city);
}
```

Cela imprime ce qui suit :

```
Tokyo
Kyoto
São Paulo
Brasília
Nairobi
Mombasa
```

Une façon de voir cela serait de dire que, pour chaque pays, nous récupérons le vecteur de ses villes, concaténons tous les vecteurs ensemble en une seule séquence et imprimons cela.

Mais rappelez-vous que les itérateurs sont paresseux : seuls les `for` appels de la boucle à la méthode de l' `flat_map` itérateur `next` provoquent l'exécution du travail. La séquence concaténée complète n'est jamais construite en mémoire. Au lieu de cela, ce que nous avons ici est une petite machine d'état qui tire de l'itérateur de ville, un élément à la fois, jusqu'à ce qu'il soit épuisé, et produit alors seulement un nouvel itérateur de ville pour le pays suivant. L'effet est celui d'une boucle imbriquée, mais emballée pour être utilisée comme itérateur.

aplatir

L' `flatten` adaptateurconcatène les éléments d'un itérateur, en supposant que chaque élément est lui-même un itérable :

```
use std:: collections::BTreeMap;

// A table mapping cities to their parks: each value is a vector.
let mut parks = BTreeMap::new();
parks.insert("Portland",  vec!["Mt. Tabor Park", "Forest Park"]);
parks.insert("Kyoto",     vec!["Tadasu-no-Mori Forest", "Maruyama Koen"]);
parks.insert("Nashville", vec!["Percy Warner Park", "Dragon Park"]);

// Build a vector of all parks. `values` gives us an iterator producing
// vectors, and then `flatten` produces each vector's elements in turn.
let all_parks:Vec<_> = parks.values().flatten().cloned().collect();

assert_eq!(all_parks,
```

```
            vec!["Tadasu-no-Mori Forest", "Maruyama Koen", "Percy Warner Pa
                 "Dragon Park", "Mt. Tabor Park", "Forest Park"]);
```

Le nom « aplatir » vient de l'image de l'aplatissement d'une structure à deux niveaux en une structure à un niveau : le `BTreeMap` et ses `Vec` s de noms sont aplatis en un itérateur produisant tous les noms.

La signature de `flatten` est la suivante :

```
fn flatten(self) -> impl Iterator<Item=Self:: Item:: Item>
    where Self:: Item:IntoIterator;
```

En d'autres termes, les éléments de l'itérateur sous-jacent doivent eux-mêmes s'implémenter `IntoIterator` pour qu'il s'agisse effectivement d'une séquence de séquences. La `flatten` méthode renvoie ensuite un itérateur sur la concaténation de ces séquences. Bien sûr, cela se fait paresseusement, en dessinant un nouvel élément `self` uniquement lorsque nous avons fini d'itérer sur le dernier.

La `flatten` méthode est utilisée de quelques manières surprenantes. Si vous avez un `Vec<Option<...>>` et que vous souhaitez itérer uniquement sur les `Some` valeurs, `flatten` fonctionne à merveille:

```
assert_eq!(vec![None, Some("day"), None, Some("one")]
           .into_iter()
           .flatten()
           .collect::<Vec<_>>(),
           vec!["day", "one"]);
```

Cela fonctionne parce que lui- `Option` même implémente `IntoIterator` , représentant une séquence de zéro ou un élément. Les `None` éléments n'apportent rien à l'itération, alors que chaque `Some` élément apporte une seule valeur. De même, vous pouvez utiliser `flatten` pour itérer sur les `Option<Vec<...>>` valeurs : `None` se comporte comme un vecteur vide.

`Result` implémente également `IntoIterator` , avec `Err` représentant une séquence vide, donc l'application `flatten` à un itérateur de `Result` valeurs extrait efficacement tous les `Err` s et les jette, ce qui entraîne un flux de valeurs de réussite non emballées. Nous ne recommandons pas d'ignorer les erreurs dans votre code, mais c'est une astuce que les gens utilisent lorsqu'ils pensent savoir ce qui se passe.

Vous pouvez vous retrouver à rechercher `flatten` ce dont vous avez réellement besoin `flat_map`. Par exemple, la méthode de la bibliothèque standard `str::to_uppercase`, qui convertit une chaîne en majuscule, fonctionne comme ceci :

```
fn to_uppercase(&self) -> String {
    self.chars()
        .map(char::to_uppercase)
        .flatten() // there's a better way
        .collect()
}
```

La raison pour laquelle `flatten` est nécessaire est que `ch.to_uppercase()` ne renvoie pas un seul caractère, mais un itérateur produisant un ou plusieurs caractères. Le mappage de chaque caractère à son équivalent majuscule donne un itérateur d'itérateurs de caractères, et le `flatten` s'occupe de les assembler tous en quelque chose que nous pouvons finalement `collect` transformer en un `String`.

Mais cette combinaison de `map` et `flatten` est si courante qu'elle `Iterator` fournit l' `flat_map` adaptateur pour ce cas précis. (En fait, `flat_map` a été ajouté à la bibliothèque standard avant `flatten`.) Ainsi, le code précédent pourrait à la place être écrit:

```
fn to_uppercase(&self) -> String {
    self.chars()
        .flat_map(char::to_uppercase)
        .collect()
}
```

prendre et prendre_pendant

Les `Iterator` traits `take` et `take_while` adaptateursvous permet de terminer une itération après un certain nombre d'éléments ou lorsqu'une fermeture décide de couper les choses. Leurs signatures sont les suivantes :

```
fn take(self, n: usize) -> impl Iterator<Item=Self:: Item>
    where Self:Sized;

fn take_while<P>(self, predicate: P) -> impl Iterator<Item=Self:: Item>
    where Self: Sized, P: FnMut(&Self:: Item) ->bool;
```

Les deux s'approprient un itérateur et renvoient un nouvel itérateur qui transmet les éléments du premier, mettant éventuellement fin à la séquence plus tôt. L' `take` itérateur revient `None` après avoir produit au maximum `n` les éléments. L' `take_while` itérateur s'applique `predicate` à chaque élément et revient `None` à la place du premier élément pour lequel `predicate` renvoie `false` et à chaque appel ultérieur à `next`.

Par exemple, étant donné un message électronique avec une ligne vide séparant les en-têtes du corps du message, vous pouvez utiliser `take_while` pour parcourir uniquement les en-têtes :

```
let message = "To: jimb\r\n\
               From: superego <editor@oreilly.com>\r\n\
               \r\n\
               Did you get any writing done today?\r\n\
               When will you stop wasting time plotting fractals?\r\n";
for header in message.lines().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}
```

Rappelez-vous de "String Literals" que lorsqu'une ligne dans une chaîne se termine par une barre oblique inverse, Rust n'inclut pas l'indentation de la ligne suivante dans la chaîne, donc aucune des lignes de la chaîne n'a d'espace blanc au début. Cela signifie que la troisième ligne de `message` est vide. L' `take_while` adaptateur termine l'itération dès qu'il voit cette ligne vide, donc ce code n'imprime que les deux lignes :

```
To: jimb
From: superego <editor@oreilly.com>
```

sauter et sauter_pendant

Les `Iterator` traits `skip` et les `skip_while` méthodessont le complément de `take` et `take_while` : ils suppriment un certain nombre d'éléments depuis le début d'une itération, ou suppriment des éléments jusqu'à ce qu'une fermeture en trouve un acceptable, puis transmettent les éléments restants tels quels. Leurs signatures sont les suivantes :

```
fn skip(self, n: usize) -> impl Iterator<Item=Self:: Item>
    where Self:Sized;

fn skip_while<P>(self, predicate: P) -> impl Iterator<Item=Self:: Item>
    where Self: Sized, P: FnMut(&Self:: Item) ->bool;
```

Une utilisation courante de l' `skip` adaptateur consiste à ignorer le nom de la commande lors de l'itération sur les arguments de ligne de commande d'un programme. Au chapitre 2 , notre calculateur de plus grand dénominateur commun a utilisé le code suivant pour parcourir ses arguments de ligne de commande :

```
for arg in std:: env::args().skip(1) {
    ...
}
```

La `std::env::args` fonctionrenvoie un itérateur qui produit les arguments du programme sous la forme `String` s, le premier élément étant le nom du programme lui-même. Ce n'est pas une chaîne que nous voulons traiter dans cette boucle. L'appel `skip(1)` à cet itérateur renvoie un nouvel itérateur qui supprime le nom du programme la première fois qu'il est appelé, puis produit tous les arguments suivants.

L' `skip_while` adaptateur utilise une fermeture pour décider du nombre d'éléments à supprimer depuis le début de la séquence. Vous pouvez parcourir les lignes du corps du message de la section précédente comme ceci :

```
for body in message.lines()
    .skip_while(|l| !l.is_empty())
    .skip(1) {
    println!("{}" , body);
}
```

Cela `skip_while` permet d'ignorer les lignes non vides, mais cet itérateur produit la ligne vide elle-même - après tout, la fermeture renvoyée `false` pour cette ligne. Nous utilisons donc `skip` également la méthode pour supprimer cela, nous donnant un itérateur dont le premier élément sera la première ligne du corps du message. Combiné avec la déclaration de `message` de la section précédente, ce code imprime :

```
Did you get any writing done today?
When will you stop wasting time plotting fractals?
```

visible

Un aperçuiterator vous permet de jeter un coup d'œil au prochain élément qui sera produit sans le consommer réellement. Vous pouvez trans-

former n'importe quel itérateur en un itérateur visible en appelant la méthode `Iterator` du trait `peekable` :

```
fn peekable(self) -> std:: iter:: Peekable<Self>
    where Self:Sized;
```

Ici, `Peekable<Self>` est a `struct` qui implémente `Iterator<Item=Self::Item>`, et `Self` est le type de l'itérateur sous-jacent.

Un `Peekable` itérateura une méthode supplémentaire `peek` quirenvoie un `Option<&Item>` : `None` si l'itérateur sous-jacent est terminé et sinon `Some(r)`, où `r` est une référence partagée à l'élément suivant. (Notez que si le type d'élément de l'itérateur est déjà une référence à quelque chose, cela finit par être une référence à une référence.)

L'appel `peek` essaie de dessiner l'élément suivant à partir de l'itérateur sous-jacent, et s'il y en a un, le met en cache jusqu'au prochain appel à `next`. Toutes les autres `Iterator` méthodes sur `Peekable` connaissent ce cache : par exemple, `iter.last()` sur un itérateur peekable `iter` sait vérifier le cache après avoir épuisé l'itérateur sous-jacent.

Les itérateurs Peekable sont essentiels lorsque vous ne pouvez pas décider du nombre d'éléments à consommer à partir d'un itérateur jusqu'à ce que vous soyez allé trop loin. Par exemple, si vous analysez des nombres à partir d'un flux de caractères, vous ne pouvez pas décider où le nombre se termine tant que vous n'avez pas vu le premier caractère non numérique qui le suit :

```
use std:: iter::Peekable;

fn parse_number<I>(tokens: &mut Peekable<I>) -> u32
    where I:Iterator<Item=char>
{
    let mut n = 0;
    loop {
        match tokens.peek() {
            Some(r) if r.is_digit(10) => {
                n = n * 10 + r.to_digit(10).unwrap();
            }
            _ => return n
        }
        tokens.next();
    }
}
```

```
let mut chars = "226153980,1766319049".chars().peekable();
assert_eq!(parse_number(&mut chars), 226153980);
// Look, `parse_number` didn't consume the comma! So we will.
assert_eq!(chars.next(), Some(','));
assert_eq!(parse_number(&mut chars), 1766319049);
assert_eq!(chars.next(), None);
```

La `parse_number` fonction utilise `peek` pour vérifier le caractère suivant et ne le consomme que s'il s'agit d'un chiffre. S'il ne s'agit pas d'un chiffre ou si l'itérateur est épuisé (c'est-à-dire si `peek` return `None`), nous renvoyons le nombre que nous avons analysé et laissons le caractère suivant dans l'itérateur, prêt à être utilisé.

fusible

Une fois an `Iterator` renvoyé `None` , le trait ne spécifie pas comment il doit se comporter si vous appelez `next` à nouveau sa méthode. La plupart des itérateurs reviennent simplement `None` , mais pas tous. Si votre code compte sur ce comportement, vous pourriez être surpris.

L' `fuse` adaptateurprend n'importe quel itérateur et en produit un qui continuera à revenir `None` une fois qu'il l'aura fait la première fois :

```
struct Flaky(bool);

impl Iterator for Flaky {
    type Item = &'static str;
    fn next(&mut self) -> Option<Self::Item> {
        if self.0 {
            self.0 = false;
            Some("totally the last item")
        } else {
            self.0 = true; // D'oh!
            None
        }
    }
}

let mut flaky = Flaky(true);
assert_eq!(flaky.next(), Some("totally the last item"));
assert_eq!(flaky.next(), None);
assert_eq!(flaky.next(), Some("totally the last item"));

let mut not_flaky = Flaky(true).fuse();
assert_eq!(not_flaky.next(), Some("totally the last item"));
assert_eq!(not_flaky.next(), None);
assert_eq!(not_flaky.next(), None);
```

L' `fuse` adaptateur est probablement plus utile dans le code générique qui doit fonctionner avec des itérateurs d'origine incertaine. Plutôt que d'espérer que chaque itérateur auquel vous devrez faire face se comportera bien, vous pouvez utiliser `fuse` pour vous en assurer.

Itérateurs réversibles et rev

Quelques itérateurssont capables de tirer des éléments des deux extrémités de la séquence. Vous pouvez inverser ces itérateurs à l'aide de l' `rev` adaptateur. Par exemple, un itérateur sur un vecteur pourrait tout aussi bien dessiner des éléments depuis la fin du vecteur que depuis le début. De tels itérateurs peuvent implémenter le `std::iter::DoubleEndedIterator` trait, qui s'étend `Iterator` :

```
trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}
```

Vous pouvez considérer un itérateur à deux extrémités comme ayant deux doigts marquant l'avant et l'arrière actuels de la séquence. Dessiner des éléments de chaque extrémité fait avancer ce doigt vers l'autre; quand les deux se rencontrent, l'itération est faite :

```
let bee_parts = ["head", "thorax", "abdomen"];

let mut iter = bee_parts.iter();
assert_eq!(iter.next(),      Some(&"head"));
assert_eq!(iter.next_back(), Some(&"abdomen"));
assert_eq!(iter.next(),      Some(&"thorax"));

assert_eq!(iter.next_back(), None);
assert_eq!(iter.next(),      None);
```

La structure d'un itérateur sur une tranche rend ce comportement facile à implémenter : c'est littéralement une paire de pointeurs vers le début et la fin de la plage d'éléments que nous n'avons pas encore produits ; `next` et `next_back` tirez simplement un élément de l'un ou de l'autre. Les itérateurs pour les collections ordonnées comme `BTreeSet` et `BTreeMap` sont également à double extrémité : leur `next_back` méthode dessine les plus grands éléments ou entrées en premier. En général, la bibliothèque standard fournit une itération double chaque fois que cela est pratique.

Mais tous les itérateurs ne peuvent pas le faire aussi facilement : un itérateur produisant des valeurs à partir d'autres threads arrivant sur un canal `Receiver` n'a aucun moyen d'anticiper quelle pourrait être la dernière valeur reçue. En général, vous devrez consulter la documentation de la bibliothèque standard pour voir quels itérateurs implémentent `DoubleEndedIterator` et lesquels ne le font pas.

Si un itérateur est à deux extrémités, vous pouvez l'inverser avec l'`rev` adaptateur :

```
fn rev(self) -> impl Iterator<Item=Self>
    where Self:Sized + DoubleEndedIterator;
```

L'itérateur renvoyé est également à double extrémité : ses méthodes `next` et sont simplement échangées : `next_back`

```
let meals = ["breakfast", "lunch", "dinner"];

let mut iter = meals.iter().rev();
assert_eq!(iter.next(), Some(&"dinner"));
assert_eq!(iter.next(), Some(&"lunch"));
assert_eq!(iter.next(), Some(&"breakfast"));
assert_eq!(iter.next(), None);
```

La plupart des adaptateurs d'itérateur, s'ils sont appliqués à un itérateur réversible, renvoient un autre itérateur réversible. Par exemple, `map` et `filter` préserver la réversibilité.

inspecter

L' `inspect` adaptateurest pratique pour déboguer les pipelines des adaptateurs d'itérateur, mais il n'est pas beaucoup utilisé dans le code de production. Il applique simplement une fermetureà une référence partagée à chaque élément, puis transmet l'élément. La fermeture ne peut pas affecter les articles, mais elle peut faire des choses comme les imprimer ou faire des affirmations à leur sujet.

Cet exemple montre un cas dans lequel la conversion d'une chaîne en majuscule modifie sa longueur :

```
let upper_case:String = "große".chars()
    .inspect(|c| println!("before: {:?}", c))
    .flat_map(|c| c.to_uppercase())
    .inspect(|c| println!(" after:     {:?}", c))
```

```
        .collect();
    assert_eq!(upper_case, "GROSSE");
```

L'équivalent majuscule de la lettre allemande minuscule "ß" est "SS", c'est pourquoi `char::to_uppercase` renvoie un itérateur sur les caractères, pas un seul caractère de remplacement. Le code précédent utilise `flat_map` pour concaténer toutes les séquences qui `to_uppercase` reviennent dans un seul `String` , en imprimant ce qui suit :

```
before: 'g'
 after:     'G'
before: 'r'
 after:     'R'
before: 'o'
 after:     'O'
before: 'ß'
 after:     'S'
 after:     'S'
before: 'e'
 after:     'E'
```

chaîne

L' `chain` adaptateurajoute un itérateur à un autre. Plus précisément, `i1.chain(i2)` renvoie un itérateur qui tire les éléments de `i1` jusqu'à ce qu'il soit épuisé, puis tire les éléments de `i2` .

La `chain` signature de l'adaptateur est la suivante :

```
fn chain<U>(self, other: U) -> impl Iterator<Item=Self:: Item>
    where Self: Sized, U: IntoIterator<Item=Self::Item>;
```

En d'autres termes, vous pouvez enchaîner un itérateur avec n'importe quel itérable qui produit le même type d'élément.

Par exemple:

```
let v:Vec<i32> = (1..4).chain([20, 30, 40]).collect();
assert_eq!(v, [1, 2, 3, 20, 30, 40]);
```

Un `chain` itérateur est réversible si ses deux itérateurs sous-jacents sont :

```
let v:Vec<i32> = (1..4).chain([20, 30, 40]).rev().collect();
assert_eq!(v, [40, 30, 20, 3, 2, 1]);
```

Un `chain` itérateur garde une trace du retour de chacun des deux itérateurs sous-jacents `None` et dirige `next` et `next_back` appelle l'un ou l'autre selon le cas.

énumérer

L' adaptateur `Iterator` du trait `enumerate` attache un index courant à la séquence, prenant un itérateur qui produit des éléments `A, B, C, ...` et renvoyant un itérateur qui produit des paires `(0, A), (1, B), (2, C), ...`. Cela semble trivial à première vue, mais il est utilisé étonnamment souvent.

Les consommateurs peuvent utiliser cet index pour distinguer un élément d'un autre et établir le contexte dans lequel traiter chacun. Par exemple, le traceur d'ensemble Mandelbrot du chapitre 2 divise l'image en huit bandes horizontales et affecte chacune à un fil différent. Ce code utilise `enumerate` pour indiquer à chaque thread à quelle partie de l'image correspond sa bande.

Cela commence par un tampon rectangulaire de pixels :

```
let mut pixels = vec![0; columns * rows];
```

Ensuite, il utilise `chunks_mut` pour diviser l'image en bandes horizontales, une par thread :

```
let threads = 8;
let band_rows = rows / threads + 1;
...
let bands:Vec<&mut [u8]> = pixels.chunks_mut(band_rows * columns).collect(
```

Et puis il itère sur les bandes, en commençant un fil pour chacune :

```
for (i, band) in bands.into_iter().enumerate() {
    let top = band_rows * i;
    // start a thread to render rows `top..top + band_rows`
    ...
}
```

Chaque itération obtient une paire `(i, band)`, où `band` est la `&mut [u8]` tranche de la mémoire tampon de pixels dans laquelle le thread doit puiser, et `i` est l'index de cette bande dans l'image globale, grâce à l'

`enumerate` adaptateur. Compte tenu des limites du tracé et de la taille des bandes, il s'agit d'informations suffisantes pour que le thread détermine quelle partie de l'image lui a été attribuée et donc dans quoi dessiner `band`.

Vous pouvez considérer les `(index, item)` paires qui `enumerate` produisent comme analogues aux `(key, value)` paires que vous obtenez lors de l'itération sur une `HashMap` ou une autre collection associative. Si vous parcourez une tranche ou un vecteur, la `index` est la « clé » sous laquelle `item` apparaît.

Zip *: français

L' `zip` adaptateurcombine deux itérateurs en un seul itérateur qui produit des paires contenant une valeur de chaque itérateur, comme une fermeture éclair joignant ses deux côtés en une seule couture. L'itérateur compressé se termine lorsque l'un des deux itérateurs sous-jacents se termine.

Par exemple, vous pouvez obtenir le même effet que l' `enumerate` adaptateuren compressant la plage de fin illimitée `0..` avec l'autre itérateur :

```
let v:Vec<_> = (0..).zip("ABCD".chars()).collect();
assert_eq!(v, vec![(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')]);
```

En ce sens, vous pouvez considérer `zip` comme une généralisation de `enumerate` : alors que `enumerate` attache des indices à la séquence, `zip` attache n'importe quel élément d'itérateur arbitraire. Nous avons suggéré précédemment que cela `enumerate` peut aider à fournir un contexte pour le traitement des éléments ; `zip` est une manière plus flexible de faire la même chose.

L'argument to `zip` n'a pas besoin d'être lui-même un itérateur ; il peut s'agir de n'importe quel itérable :

```
use std:: iter::repeat;

let endings = ["once", "twice", "chicken soup with rice"];
let rhyme:Vec<_> = repeat("going")
    .zip(endings)
    .collect();
assert_eq!(rhyme, vec![("going", "once"),
                       ("going", "twice"),
                       ("going", "chicken soup with rice")]);
```

by_ref

Tout au long de cette section, nous avons attaché des adaptateurs aux itérateurs. Une fois que vous l'avez fait, pouvez-vous retirer l'adaptateur ? Généralement, non : les adaptateurs s'approprient l'itérateur sous-jacent et ne fournissent aucune méthode pour le rendre.

`by_ref` La méthode d'un itérateuremprunte une référence mutable à l'itérateur afin que vous puissiez appliquer des adaptateurs à la référence. Lorsque vous avez fini de consommer des éléments de ces adaptateurs, vous les supprimez, l'emprunt prend fin et vous retrouvez l'accès à votre itérateur d'origine.

Par exemple, plus tôt dans le chapitre, nous avons montré commentpour utiliser `take_while` et `skip_while` traiter les lignes d'en-tête et le corps d'un message électronique. Mais que se passe-t-il si vous voulez faire les deux, en utilisant le même itérateur sous-jacent ? En utilisant `by_ref` , nous pouvons utiliser `take_while` pour gérer les en-têtes, et lorsque cela est fait, récupérer l'itérateur sous-jacent, qui `take_while` est resté exactement en position pour gérer le corps du message :

```
let message = "To: jimb\r\n\
               From: id\r\n\
               \r\n\
               Oooooh, donuts!!\r\n";

let mut lines = message.lines();

println!("Headers:");
for header in lines.by_ref().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}

println!("\nBody:");
for body in lines {
    println!("{}" , body);
}
```

L'appel `lines.by_ref()` emprunte une référence mutable à l'itérateur, et c'est cette référence dont l' `take_while` itérateur s'approprie. Cet itérateur sort de la portée à la fin de la première `for` boucle, ce qui signifie que l'emprunt est terminé, vous pouvez donc l'utiliser `lines` à nouveau dans la deuxième `for` boucle. Cela imprime ce qui suit :

```
Headers:
To: jimb
From: id

Body:
Oooooh, donuts!!
```

La `by_ref` définition de l'adaptateur est triviale : elle renvoie une référence mutable à l'itérateur. Ensuite, la bibliothèque standard inclut cette étrange petite implémentation :

```
impl<'a, I: Iterator + ?Sized> Iterator for &'a mut I {
    type Item = I:: Item;
    fn next(&mut self) -> Option<I:: Item> {
        (**self).next()
    }
    fn size_hint(&self) ->(usize, Option<usize>) {
        (**self).size_hint()
    }
}
```

En d'autres termes, si `I` est un type d'itérateur, alors `&mut I` est aussi un itérateur, dont les méthodes `next` et `size_hint` s'en remettre à son référent. Lorsque vous appelez un adaptateur sur une référence mutable à un itérateur, l'adaptateur s'approprie la *référence* , et non l'itérateur lui-même. C'est juste un emprunt qui se termine lorsque l'adaptateur sort de la portée.

cloné, copié

L' `cloned` adaptateurprend un itérateur qui produit des références et renvoie un itérateur qui produit des valeurs clonées à partir de ces références, un peu comme `iter.map(|item| item.clone())` . Naturellement, le type référent doit implémenter `Clone` . Par exemple:

```
let a = ['1', '2', '3', '∞'];

assert_eq!(a.iter().next(),          Some(&'1'));
assert_eq!(a.iter().cloned().next(), Some('1'));
```

L' `copied` adaptateurest la même idée, mais plus restrictive : le type référent doit implémenter `Copy` . Un appel comme `iter.copied()` est à peu près le même que `iter.map(|r| *r)` . Étant donné que chaque type qui implémente `Copy` également implémente `Clone` , `cloned` est stricte-

ment plus général, mais selon le type d'élément, un `clone` appel peut effectuer des quantités arbitraires d'allocation et de copie. Si vous supposez que cela ne se produira jamais parce que votre type d'élément est quelque chose de simple, il est préférable de l'utiliser `copied` pour que le vérificateur de type vérifie vos hypothèses.

cycle

L' `cycle` adaptateurrenvoie un itérateur qui répète indéfiniment la séquence produite par l'itérateur sous-jacent. L'itérateur sous-jacent doit être implémenté `std::clone::Clone` pour `cycle` pouvoir sauvegarder son état initial et le réutiliser à chaque redémarrage du cycle.

Par exemple:

```
let dirs = ["North", "East", "South", "West"];
let mut spin = dirs.iter().cycle();
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
assert_eq!(spin.next(), Some(&"South"));
assert_eq!(spin.next(), Some(&"West"));
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
```

Ou, pour une utilisation vraiment gratuite des itérateurs :

```
use std:: iter::{once, repeat};

let fizzes = repeat("").take(2).chain(once("fizz")).cycle();
let buzzes = repeat("").take(4).chain(once("buzz")).cycle();
let fizzes_buzzes = fizzes.zip(buzzes);

let fizz_buzz = (1..100).zip(fizzes_buzzes)
    .map(|tuple|
         match tuple {
             (i, ("", "")) => i.to_string(),
             (_, (fizz, buzz)) => format!("{}{}", fizz, buzz)
         });

for line in fizz_buzz {
    println!("{}", line);
}
```

Cela joue un jeu de mots pour enfants, maintenant parfois utilisé comme question d'entretien d'embauche pour les codeurs, dans lequel les joueurs comptent à tour de rôle, remplaçant tout nombre divisible par trois par le

mot `fizz`, et tout nombre divisible par cinq par `buzz`. Nombresdivisible par les deux devenir `fizzbuzz`.

Consommer des itérateurs

Jusqu'à présent, nous avons couvert la création d'itérateurs et leur adaptation dans de nouveaux itérateurs ; ici, nous terminons le processus en montrant des façons de consommerleur.

Bien sûr, vous pouvez utiliser un itérateur avec une `for` boucle ou l'appeler `next` explicitement, mais il existe de nombreuses tâches courantes que vous ne devriez pas avoir à écrire encore et encore. Le `Iterator` trait fournit une large sélection de méthodes pour couvrir bon nombre d'entre elles.

Accumulation simple : compte, somme, produit

La `count` méthodetire les éléments d'un itérateur jusqu'à ce qu'il revienne `None` et vous indique combien il en a. Voici un programme court qui compte le nombre de lignes sur son entrée standard :

```
use std:: io:: prelude::*;

fn main() {
    let stdin = std:: io::stdin();
    println!("{}", stdin.lock().lines().count());
}
```

Les méthodes `sum` et `product` calcule la somme ou le produit des éléments de l'itérateur, qui doivent être des entiers ou des nombres à virgule flottante :

```
fn triangle(n: u64) ->u64 {
    (1..=n).sum()
}
assert_eq!(triangle(20), 210);

fn factorial(n: u64) ->u64 {
    (1..=n).product()
}
assert_eq!(factorial(20), 2432902008176640000);
```

(Vous pouvez étendre `sum` et `product` travailler avec d'autres types en implémentant les traits `std::iter::Sum` et `std::iter::Product`, que

nous ne décrirons pas dans ce livre.)

maximum minimum

Les méthodes `min` et `max` en `Iterator` retour, le plus petit ou le plus grand élément produit par l'itérateur. Le type d'élément de l'itérateur doit être implémenté `std::cmp::Ord` afin que les éléments puissent être comparés les uns aux autres. Par exemple:

```
assert_eq!([-2, 0, 1, 0, -2, -5].iter().max(), Some(&1));
assert_eq!([-2, 0, 1, 0, -2, -5].iter().min(), Some(&-5));
```

Ces méthodes renvoient un `Option<Self::Item>` afin qu'elles puissent revenir `None` si l'itérateur ne produit aucun élément.

Comme expliqué dans "Comparaisons d'équivalence" , la virgule flottante de Rusttypes `f32` et `f64` implémentez uniquement `std::cmp::PartialOrd` , pas `std::cmp::Ord` , vous ne pouvez donc pas utiliser les méthodes `min` et `max` pour calculer le plus petit ou le plus grand d'une séquence de nombres à virgule flottante. Ce n'est pas un aspect populaire de la conception de Rust, mais c'est délibéré : il n'est pas clair ce que ces fonctions devraient faire avec les valeurs IEEE NaN. Les ignorer simplement risquerait de masquer des problèmes plus graves dans le code.

Si vous savez comment vous souhaitez gérer les valeurs NaN, vous pouvez utiliser les méthodes d'itération `max_by` et à la `min_by` place, qui vous permettent de fournir votre propre fonction de comparaison.

max_by, min_by

Les méthodes `max_by` et `min_by` renvoie l'élément maximum ou minimum produit par l'itérateur, tel que déterminé par une fonction de comparaison que vous fournissez :

```
use std:: cmp::Ordering;

// Compare two f64 values. Panic if given a NaN.
fn cmp(lhs: &f64, rhs: &f64) ->Ordering {
    lhs.partial_cmp(rhs).unwrap()
}

let numbers = [1.0, 4.0, 2.0];
assert_eq!(numbers.iter().copied().max_by(cmp), Some(4.0));
```

```
assert_eq!(numbers.iter().copied().min_by(cmp), Some(1.0));

let numbers = [1.0, 4.0, std:: f64::NAN, 2.0];
assert_eq!(numbers.iter().copied().max_by(cmp), Some(4.0)); // panics
```

Les méthodes `max_by` et `min_by` transmettent les éléments à la fonction de comparaison par référence afin qu'ils puissent fonctionner efficacement avec n'importe quel type d'itérateur, donc `cmp` s'attend à prendre ses arguments par référence, même si nous avions l'habitude `copied` d'obtenir un itérateur qui produit des `f64` éléments.

max_by_key, min_by_key

Les méthodes `max_by_key` et `min_by_key` on `Iterator` vous permet de sélectionner l'élément maximum ou minimum déterminé par une fermeture appliquée à chaque élément. La fermeture peut sélectionner un champ de l'élément ou effectuer un calcul sur les éléments. Étant donné que vous êtes souvent intéressé par les données associées à un minimum ou à un maximum, pas seulement à l'extremum lui-même, ces fonctions sont souvent plus utiles que `min` et `max` . Leurs signatures sont les suivantes :

```
fn min_by_key<B: Ord, F>(self, f: F) -> Option<Self:: Item>
    where Self: Sized, F: FnMut(&Self:: Item) ->B;

fn max_by_key<B: Ord, F>(self, f: F) -> Option<Self:: Item>
    where Self: Sized, F: FnMut(&Self:: Item) ->B;
```

Autrement dit, étant donné une fermeture qui prend un élément et renvoie n'importe quel type ordonné `B` , renvoie l'élément pour lequel la fermeture a renvoyé le maximum ou le minimum `B` , ou `None` si aucun élément n'a été produit.

Par exemple, si vous devez parcourir une table de hachage de villes pour trouver les villes les plus peuplées et les plus peuplées, vous pouvez écrire :

```
use std:: collections::HashMap;

let mut populations = HashMap::new();
populations.insert("Portland",  583_776);
populations.insert("Fossil",        449);
populations.insert("Greenhorn",       2);
populations.insert("Boring",      7_762);
populations.insert("The Dalles", 15_340);
```

```
assert_eq!(populations.iter().max_by_key(|&(_name, pop)| pop),
           Some((&"Portland", &583_776)));
assert_eq!(populations.iter().min_by_key(|&(_name, pop)| pop),
           Some((&"Greenhorn", &2)));
```

La fermeture `|&(_name, pop)| pop` est appliquée à chaque élément produit par l'itérateur et renvoie la valeur à utiliser pour la comparaison, dans ce cas, la population de la ville. La valeur renvoyée est l'élément entier, pas seulement la valeur renvoyée par la fermeture. (Naturellement, si vous faisiez souvent des requêtes comme celle-ci, vous voudriez probablement trouver un moyen plus efficace de trouver les entrées que de faire une recherche linéaire dans la table.)

Comparer des séquences d'articles

Vous pouvez utiliser les opérateurs `<` et pour comparer `==` chaînes, vecteurs et tranches, en supposant que leurs éléments individuels peuvent être comparés. Bien que les itérateurs ne prennent pas en charge les opérateurs de comparaison de Rust, ils fournissent des méthodes comme `eq` et `lt` qui font le même travail, tirant des paires d'éléments des itérateurs et les comparant jusqu'à ce qu'une décision puisse être prise. Par exemple:

```
let packed =  "Helen of Troy";
let spaced =  "Helen   of    Troy";
let obscure = "Helen of Sandusky"; // nice person, just not famous

assert!(packed != spaced);
assert!(packed.split_whitespace().eq(spaced.split_whitespace()));

// This is true because ' ' < 'o'.
assert!(spaced < obscure);

// This is true because 'Troy' > 'Sandusky'.
assert!(spaced.split_whitespace().gt(obscure.split_whitespace()));
```

Les appels pour `split_whitespace` renvoyer des itérateurs sur les mots séparés par des espaces de la chaîne. L'utilisation des méthodes `eq` et `gt` sur ces itérateurs effectue une comparaison mot par mot, au lieu d'une comparaison caractère par caractère. Tout cela est possible car `&str` implémente `PartialOrd` et `PartialEq`.

Les itérateurs fournissent les méthodes `eq` et `ne` pour les comparaisons d'égalité , et les méthodes `lt`, `le`, `gt` et `ge` pour les comparaisons or-

données. Les méthodes `cmp` et `partial_cmp` se comportent comme les méthodes correspondantes des traits `Ord` et . `PartialOrd`

tout et tout

Les méthodes `any` et `all` applique une fermeture à chaque élément produit par l'itérateur et retourne `true` si la fermeture revient `true` pour n'importe quel élément, ou pour tous les éléments :

```
let id = "Iterator";

assert!( id.chars().any(char:: is_uppercase));
assert!(!id.chars().all(char::is_uppercase));
```

Ces méthodes ne consomment que le nombre d'éléments nécessaires pour déterminer la réponse. Par exemple, si la fermeture revient `true` pour un élément donné, elle `any` revient `true` immédiatement, sans tirer d'autres éléments de l'itérateur.

position, rposition et ExactSizeIterator

La `position` méthodeapplique une fermeture à chaque élément de l'itérateur et renvoie l'index du premier élément pour lequel la fermeture renvoie `true` . Plus précisément, il renvoie un `Option` de l'index : si la fermeture ne renvoie `true` aucun élément, `position` renvoie `None` . Il arrête de dessiner des éléments dès que la fermeture revient `true` . Par exemple:

```
let text = "Xerxes";
assert_eq!(text.chars().position(|c| c == 'e'), Some(1));
assert_eq!(text.chars().position(|c| c == 'z'), None);
```

La `rposition` méthodeest le même, sauf qu'il recherche à partir de la droite. Par exemple :

```
let bytes = b"Xerxes";
assert_eq!(bytes.iter().rposition(|&c| c == b'e'), Some(4));
assert_eq!(bytes.iter().rposition(|&c| c == b'X'), Some(0));
```

La `rposition` méthode nécessite un itérateur réversible afin de pouvoir dessiner des éléments à partir de l'extrémité droite de la séquence. Il nécessite également un itérateur de taille exacte afin qu'il puisse affecter des index de la même manière `position` , en commençant par `0` à

gauche. Un itérateur de taille exacte est celui qui implémente le `std::iter::ExactSizeIterator` trait:

```
trait ExactSizeIterator: Iterator {
    fn len(&self) -> usize { ... }
    fn is_empty(&self) ->bool { ... }
}
```

La `len` méthoderenvoie le nombre d'éléments restants et la `is_empty` méthode renvoie `true` si l'itération est terminée.

Naturellement, tous les itérateurs ne savent pas à l'avance combien d'éléments ils produiront. Par exemple, l' `str::chars` itérateur utilisé précédemment ne le fait pas (puisque UTF-8 est un encodage à largeur variable), vous ne pouvez donc pas l'utiliser `rposition` sur des chaînes. Mais un itérateur sur un tableau d'octets connaît certainement la longueur du tableau, il peut donc implémenter `ExactSizeIterator` .

plier et replier

La `fold` méthodeest un outil très général pour accumuler une sorte de résultat sur toute la séquence d'éléments produits par un itérateur. Étant donné une valeur initiale, que nous appellerons l' *accumulateur*, et une fermeture, `fold` applique à plusieurs reprises la fermeture à l'accumulateur actuel et à l'élément suivant de l'itérateur. La valeur renvoyée par la fermeture est prise comme nouvel accumulateur, à transmettre à la fermeture avec l'élément suivant. La valeur finale de l'accumulateur est ce qui `fold` lui-même renvoie. Si la séquence est vide, `fold` renvoie simplement l'accumulateur initial.

De nombreuses autres méthodes de consommation des valeurs d'un itérateur peuvent être écrites comme des utilisations de `fold` :

```
let a = [5, 6, 7, 8, 9, 10];

assert_eq!(a.iter().fold(0, |n, _| n+1), 6);          // count
assert_eq!(a.iter().fold(0, |n, i| n+i), 45);         // sum
assert_eq!(a.iter().fold(1, |n, i| n*i), 151200);     // product

// max
assert_eq!(a.iter().cloned().fold(i32:: min_value(), std:: cmp::max),
           10);
```

La `fold` signature de la méthode est la suivante :

```
fn fold<A, F>(self, init: A, f: F) -> A
    where Self: Sized, F: FnMut(A, Self::Item) ->A;
```

Ici, `A` c'est le type d'accumulateur. L' `init` argument est un `A`, tout comme le premier argument et la valeur de retour de la fermeture, et la valeur de retour d' elle- `fold` même.

Notez que les valeurs d'accumulateur sont déplacées vers l'intérieur et l'extérieur de la fermeture, vous pouvez donc les utiliser `fold` avec `Copy` des types sans accumulateur :

```
let a = ["Pack", "my", "box", "with",
         "five", "dozen", "liquor", "jugs"];

// See also: the `join` method on slices, which won't
// give you that extra space at the end.
let pangram = a.iter()
    .fold(String::new(), |s, w| s + w + " ");
assert_eq!(pangram, "Pack my box with five dozen liquor jugs ");
```

La `rfold` méthodeest identique à `fold`, sauf qu'il nécessite un itérateur à deux extrémités et traite ses éléments du dernier au premier :

```
let weird_pangram = a.iter()
    .rfold(String::new(), |s, w| s + w + " ");
assert_eq!(weird_pangram, "jugs liquor dozen five with box my Pack ");
```

try_fold et try_rfold

La `try_fold` méthodeest identique à `fold`, sauf que l'itération peut se terminer plus tôt, sans consommer toutes les valeurs de l'itérateur. La valeur renvoyée par la fermeture à laquelle vous passez `try_fold` indique si elle doit revenir immédiatement ou continuer à replier les éléments de l'itérateur.

Votre fermeture peut renvoyer n'importe lequel de plusieurs types, indiquant comment le pliage doit se dérouler :

- Si votre fermeture renvoie `Result<T, E>`, peut-être parce qu'elle effectue des E/S ou effectue une autre opération faillible, alors le retour `Ok(v)` indique `try_fold` de continuer le pliage, avec `v` comme nouvelle valeur d'accumulateur. Le retour `Err(e)` provoque l'arrêt im-

médiat du pliage. La valeur finale du repli est a `Result` portant la valeur finale de l'accumulateur, ou l'erreur renvoyée par la fermeture.

- Si votre fermeture renvoie `Option<T>`, `Some(v)` indique alors que le pliage doit continuer avec `v` comme nouvelle valeur d'accumulateur et `None` indique que l'itération doit s'arrêter immédiatement. La valeur finale du pli est également un `Option`.
- Enfin, la fermeture peut renvoyer une `std::ops::ControlFlow` valeur. Ce type est une énumération avec deux variantes, `Continue(c)` et `Break(b)`, signifiant continuer avec une nouvelle valeur d'accumulateur `c` ou s'arrêter plus tôt. Le résultat du pli est une `ControlFlow` valeur : `Continue(v)` si le pli a consommé tout l'itérateur, donnant la valeur finale de l'accumulateur `v` ; ou `Break(b)`, si la fermeture a renvoyé cette valeur.
  `Continue(c)` et `Break(b)` se comportent exactement comme `Ok(c)` et `Err(b)`. L'avantage d'utiliser à la `ControlFlow` place de `Result` est que cela rend votre code un peu plus lisible lorsqu'une sortie anticipée n'indique pas une erreur, mais simplement que la réponse est prête plus tôt. Nous en montrons un exemple ci-dessous.

Voici un programme qui additionne les nombres lus à partir de son entrée standard :

```
use std:: error:: Error;
use std:: io:: prelude:: *;
use std:: str::FromStr;

fn main() -> Result<(), Box<dyn Error>> {
    let stdin = std:: io:: stdin();
    let sum = stdin.lock()
        .lines()
        .try_fold(0, |sum, line| -> Result<u64, Box<dyn Error>> {
            Ok(sum + u64::from_str(&line?.trim())?)
        })?;
    println!("{}", sum);
    Ok(())
}
```

L' `lines` itérateursur les flux d'entrée mis en mémoire tampon produit des éléments de type `Result<String, std::io::Error>`, et l'analyse `String` en tant qu'entier peut également échouer. L'utilisation `try_fold` ici permet à la fermeture de revenir `Result<u64, ...>`, nous pouvons donc utiliser l' `?` opérateur pour propager les échecs de la fermeture à la `main` fonction.

Parce `try_fold` qu'il est si flexible, il est utilisé pour implémenter de nombreuses `Iterator` autres méthodes grand public de . Par exemple, voici une implémentation de `all` :

```
fn all<P>(&mut self, mut predicate: P) -> bool
    where P: FnMut(Self::Item) -> bool,
          Self: Sized
{
    use std::ops::ControlFlow::*;
    self.try_fold((), |_, item| {
        if predicate(item) { Continue(()) } else { Break(()) }
    }) == Continue(())
}
```

Notez que cela ne peut pas être écrit avec ordinaire `fold` : `all` promet d'arrêter de consommer les éléments de l'itérateur sous-jacent dès qu'il `predicate` renvoie false, mais `fold` consomme toujours l'itérateur entier.

Si vous implémentez votre propre type d'itérateur, il est utile de déterminer si votre itérateur pourrait être implémenté `try_fold` plus efficacement que la définition par défaut du `Iterator` trait. Si vous pouvez accélérer `try_fold` , toutes les autres méthodes construites dessus en bénéficieront également.

La `try_rfold` méthode, comme son nom l'indique, est la même que `try_fold` , sauf qu'elle tire les valeurs de l'arrière, au lieu de l'avant, et nécessite un itérateur à double extrémité.

nième, nième_retour

La `nth` méthodeprend un index `n` , ignore autant d'éléments de l'itérateur et renvoie l'élément suivant, ou `None` si la séquence se termine avant ce point. L'appel `.nth(0)` est équivalent à `.next()` .

Il ne s'approprie pas l'itérateur comme le ferait un adaptateur, vous pouvez donc l'appeler plusieurs fois :

```
let mut squares = (0..10).map(|i| i*i);

assert_eq!(squares.nth(4), Some(16));
assert_eq!(squares.nth(0), Some(25));
assert_eq!(squares.nth(6), None);
```

Sa signature est montrée ici :

```
fn nth(&mut self, n: usize) -> Option<Self:: Item>
    where Self:Sized;
```

La `nth_back` méthode est sensiblement la même, sauf qu'elle tire de l'arrière d'un itérateur à double extrémité. L'appel `.nth_back(0)` est équivalent à `.next_back()` : il renvoie le dernier élément, ou `None` si l'itérateur est vide.

dernière

La `last` méthoderenvoie le dernier élément produit par l'itérateur, ou `None` s'il est vide. Sa signature est la suivante :

```
fn last(self) -> Option<Self::Item>;
```

Par exemple:

```
let squares = (0..10).map(|i| i*i);
assert_eq!(squares.last(), Some(81));
```

Cela consomme tous les éléments de l'itérateur en commençant par le début, même si l'itérateur est réversible. Si vous avez un itérateur réversible et que vous n'avez pas besoin de consommer tous ses éléments, vous devriez plutôt simplement écrire `iter.next_back()` .

find, rfind et find_map

La `find` méthodetire des éléments d'un itérateur, renvoyant le premier élément pour lequel la fermeture donnée renvoie `true` , ou `None` si la séquence se termine avant qu'un élément approprié ne soit trouvé. Sa signature est :

```
fn find<P>(&mut self, predicate: P) -> Option<Self:: Item>
    where Self: Sized,
          P: FnMut(&Self:: Item) ->bool;
```

La `rfind` méthodeest similaire, mais il nécessite un itérateur à deux extrémités et recherche les valeurs de l'arrière vers l'avant, renvoyant le *dernier* élément pour lequel la fermeture renvoie `true` .

Par exemple, en utilisant le tableau des villes et des populations de « max_by_key, min_by_key » , vous pouvez écrire :

```
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 1_000_000),
           None);
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 500_000),
           Some((&"Portland", &583_776)));
```

Aucune des villes du tableau n'a une population supérieure à un million, mais il y a une ville avec un demi-million d'habitants.

Parfois, votre clôture n'est pas qu'un simple prédicat jetant un jugement booléen sur chaque élément et passant à autre chose : il peut s'agir de quelque chose de plus complexe qui produit une valeur intéressante en soi. Dans ce cas, `find_map` est exactement ce que vous voulez. Sa signature est :

```
fn find_map<B, F>(&mut self, f: F) -> Option<B> where
    F: FnMut(Self:: Item) ->Option<B>;
```

C'est comme `find` , sauf qu'au lieu de retourner `bool` , la fermeture devrait retourner un `Option` d'une certaine valeur. `find_map` renvoie le premier `Option` qui est `Some` .

Par exemple, si nous avons une base de données des parcs de chaque ville, nous pourrions vouloir voir si certains d'entre eux sont des volcans et fournir le nom du parc si c'est le cas :

```
let big_city_with_volcano_park = populations.iter()
    .find_map(|(&city, _)| {
        if let Some(park) = find_volcano_park(city, &parks) {
            // find_map returns this value, so our caller knows
            // *which* park we found.
            return Some((city, park.name));
        }

        // Reject this item, and continue the search.
        None
    });

assert_eq!(big_city_with_volcano_park,
           Some(("Portland", "Mt. Tabor Park")));
```

Création de collections : collect et FromIterator

Tout au long du livre, nous avons utilisé la `collect` méthodepour construire des vecteurs contenant les éléments d'un itérateur. Par exemple, au chapitre 2 , nous avons appelé `std::env::args()` pour obtenir un itérateur sur les arguments de la ligne de commande du programme, puis avons appelé la `collect` méthode de cet itérateur pour les rassembler dans un vecteur :

```
let args: Vec<String> = std:: env::args().collect();
```

Mais `collect` n'est pas spécifique aux vecteurs : en fait, il peut créer n'importe quel type de collection à partir de la bibliothèque standard de Rust, tant que l'itérateur produit un type d'élément approprié :

```
use std:: collections::{HashSet, BTreeSet, LinkedList, HashMap, BTreeMap};

let args: HashSet<String> = std:: env:: args().collect();
let args: BTreeSet<String> = std:: env:: args().collect();
let args: LinkedList<String> = std:: env::args().collect();

// Collecting a map requires (key, value) pairs, so for this example,
// zip the sequence of strings with a sequence of integers.
let args: HashMap<String, usize> = std:: env:: args().zip(0..).collect();
let args: BTreeMap<String, usize> = std:: env::args().zip(0..).collect();

// and so on
```

Naturellement, `collect` lui-même ne sait pas construire tous ces types. Au lieu de cela, lorsqu'un type de collection aime `Vec` ou `HashMap` sait se construire à partir d'un itérateur, il implémente le `std::iter::FromIterator` trait, pour qui `collect` n'est qu'un placage commode :

```
trait FromIterator<A>: Sized {
    fn from_iter<T: IntoIterator<Item=A>>(iter: T) ->Self;
}
```

Si un type de collection implémente `FromIterator<A>` , alors sa fonction associée au type `from_iter` construit une valeur de ce type à partir d'un itérable produisant des éléments de type `A` .

Dans le cas le plus simple, l'implémentation pourrait simplement construire une collection vide, puis ajouter les éléments de l'itérateur un par un. Par exemple, `std::collections::LinkedList` la mise en œuvre de `FromIterator` fonctionne de cette façon.

Cependant, certains types peuvent faire mieux que cela. Par exemple, construire un vecteur à partir d'un itérateur `iter` pourrait être aussi simple que :

```
let mut vec = Vec::new();
for item in iter {
    vec.push(item)
}
vec
```

Mais ce n'est pas idéal : à mesure que le vecteur grandit, il peut avoir besoin d'étendre son tampon, ce qui nécessite un appel à l'allocateur de tas et une copie des éléments existants. Les vecteurs prennent des mesures algorithmiques pour maintenir cette surcharge faible, mais s'il y avait un moyen d'allouer simplement un tampon de la bonne taille pour commencer, il n'y aurait aucun besoin de redimensionner.

C'est là qu'intervient la méthode `Iterator` du trait : `size_hint`

```
trait Iterator {
    ...
    fn size_hint(&self) ->(usize, Option<usize>) {
        (0, None)
    }
}
```

Cette méthode renvoie une limite inférieure et une limite supérieure facultative sur le nombre d'éléments que l'itérateur produira. La définition par défaut renvoie zéro comme limite inférieure et refuse de nommer une limite supérieure, en disant, en fait, "je n'en ai aucune idée", mais de nombreux itérateurs peuvent faire mieux que cela. Un itérateur sur a `Range` , par exemple, sait exactement combien d'éléments il produira, tout comme un itérateur sur a `Vec` ou `HashMap` . Ces itérateurs fournissent leurs propres définitions spécialisées pour `size_hint` .

Ces limites sont exactement les informations dont `Vec` l'implémentation a `FromIterator` besoin pour dimensionner correctement le tampon du nouveau vecteur dès le départ. Les insertions vérifient toujours que le tampon est suffisamment grand, donc même si l'indice est incorrect, seules les performances sont affectées, pas la sécurité. D'autres types peuvent suivre des étapes similaires : par exemple, `HashSet` et `HashMap` également utiliser `Iterator::size_hint` pour choisirune taille initiale appropriée pour leur table de hachage.

Une remarque à propos de l'inférence de type : en haut de cette section, il est un peu étrange de voir le même appel, `std::env::args().collect()`, produire quatre types de collections différents en fonction de son contexte. Le type de retour de `collect` est son paramètre de type, donc les deux premiers appels sont équivalents à ce qui suit :

```
let args = std:: env:: args().collect:: <Vec<String>>();
let args = std:: env:: args().collect::<HashSet<String>>();
```

Mais tant qu'il n'y a qu'un seul type qui pourrait éventuellement fonctionner comme `collect` argument de , l'inférence de type de Rust le fournira pour vous. Lorsque vous épelez le type de `args`, vous vous assurez que c'est le cas.

Le trait d'extension

Si un type implémente le `std::iter::Extend` trait, alors sa `extend` méthode ajoute les éléments d'un itérable à la collection :

```
let mut v:Vec<i32> = (0..5).map(|i| 1 << i).collect();
v.extend([31, 57, 99, 163]);
assert_eq!(v, [1, 2, 4, 8, 16, 31, 57, 99, 163]);
```

Toutes les collections standard implémentent `Extend`, donc elles ont toutes cette méthode ; tout comme `String`. Les tableaux et les tranches, qui ont une longueur fixe, n'en ont pas.

La définition du trait est la suivante :

```
trait Extend<A> {
    fn extend<T>(&mut self, iter: T)
        where T:IntoIterator<Item=A>;
}
```

Évidemment, cela ressemble beaucoup à `std::iter::FromIterator` : qui crée une nouvelle collection, alors qu'il `Extend` étend une collection existante. En effet, plusieurs implémentationsde `FromIterator` dans la bibliothèque standard, créez simplement une nouvelle collection vide, puis appelez- `extend` la pour la remplir. Par exemple, l'implémentation de `FromIterator` for `std::collections::LinkedList` fonctionne de la manière suivante :

```
impl<T> FromIterator<T> for LinkedList<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut list = Self::new();
        list.extend(iter);
        list
    }
}
```

cloison

La `partition` méthodedivise les éléments d'un itérateur entre deux collections, en utilisant une fermeture pour décider où chaque élément appartient :

```
let things = ["doorknob", "mushroom", "noodle", "giraffe", "grapefruit"];

// Amazing fact: the name of a living thing always starts with an
// odd-numbered letter.
let (living, nonliving):(Vec<&str>, Vec<&str>)
    = things.iter().partition(|name| name.as_bytes()[0] & 1 != 0);

assert_eq!(living,    vec!["mushroom", "giraffe", "grapefruit"]);
assert_eq!(nonliving, vec!["doorknob", "noodle"]);
```

Comme `collect` , `partition` peutfaites toutes sortes de collections que vous aimez, bien que les deux doivent être du même type. Et comme `collect` , vous devrez spécifier le type de retour : l'exemple précédent écrit le type de `living` et `nonliving` et laisse l'inférence de type choisir les bons paramètres de type pour l'appel à `partition` .

La signature de `partition` est la suivante :

```
fn partition<B, F>(self, f: F) -> (B, B)
    where Self: Sized,
          B: Default + Extend<Self:: Item>,
          F: FnMut(&Self:: Item) ->bool;
```

Alors que `collect` requiert son type de résultat pour implémenter `FromIterator` , `partition` requiert à la place `std::default::Default` , ce quitoutes les collections Rust implémentent en retournant une collection vide, et `std::default::Extend` .

D'autres langages proposent des `partition` opérations qui divisent simplement l'itérateur en deux itérateurs, au lieu de créer deux collections.

Mais ce n'est pas un bon choix pour Rust : les éléments tirés de l'itérateur sous-jacent mais pas encore tirés de l'itérateur partitionné approprié devraient être mis en mémoire tampon quelque part ; vous finiriez par constituer une collection quelconque en interne, de toute façon.

for_each et try_for_each

La `for_each` méthodeapplique simplement une fermeture à chaque élément :

```
["doves", "hens", "birds"].iter()
    .zip(["turtle", "french", "calling"])
    .zip(2..5)
    .rev()
    .map(|((item, kind), quantity)| {
        format!("{} {} {}", quantity, kind, item)
    })
    .for_each(|gift| {
        println!("You have received: {}", gift);
    });
```

Cela imprime :

```
You have received: 4 calling birds
You have received: 3 french hens
You have received: 2 turtle doves
```

`for` Ceci est très similaire à une boucle simple , dans laquelle vous pouvez également utiliser des structures de contrôle telles que `break` et `continue` . Mais les longues chaînes d'appels d'adaptateur comme celui-ci sont un peu gênantes dans les `for` boucles :

```
for gift in ["doves", "hens", "birds"].iter()
    .zip(["turtle", "french", "calling"])
    .zip(2..5)
    .rev()
    .map(|((item, kind), quantity)| {
        format!("{} {} {}", quantity, kind, item)
    })
{
    println!("You have received: {}", gift);
}
```

Le motif étant lié, `gift` , peut se retrouver assez loin du corps de boucle dans lequel il est utilisé.

Si votre fermeture doit être faillible ou sortir plus tôt, vous pouvez utiliser `try_for_each` :

```
...
    .try_for_each(|gift| {
        writeln!(&mut output_file, "You have received: {}", gift)
    })?;
```

Implémentation de vos propres itérateurs

Vous pouvez mettre en œuvreles `IntoIterator` et `Iterator` traitspour vos propres types, ce qui rend tous les adaptateurs et consommateurs présentés dans ce chapitre disponibles, ainsi que de nombreux autres codes de bibliothèque et de crate écrits pour fonctionner avec l'interface d'itérateur standard. Dans cette section, nous allons montrer un itérateur simple sur un type de plage, puis un itérateur plus complexe sur un type d'arbre binaire.

Supposons que nous ayons le type de plage suivant (simplifié à partir du type de bibliothèque standard `std::ops::Range<T>`) :

```
struct I32Range {
    start: i32,
    end:i32
}
```

L'itération sur an `I32Range` nécessite deux éléments d'état : la valeur actuelle et la limite à laquelle l'itération doit se terminer. Cela se trouve être un bon ajustement pour le `I32Range` type lui-même, en utilisant `start` comme valeur suivante et `end` comme limite. Vous pouvez donc implémenter `Iterator` comme ceci:

```
impl Iterator for I32Range {
    type Item = i32;
    fn next(&mut self) ->Option<i32> {
        if self.start >= self.end {
            return None;
        }
        let result = Some(self.start);
        self.start += 1;
        result
```

```
        }
    }
```

Cet itérateur produit des `i32` éléments, c'est donc le `Item` type. Si l'itération est terminée, `next` renvoie `None` ; sinon, il produit la valeur suivante et met à jour son état pour se préparer au prochain appel.

Bien sûr, une `for` boucle utilise `IntoIterator::into_iter` pour convertir son opérande en itérateur. Mais la bibliothèque standard fournit une implémentation globale de `IntoIterator` pour chaque type qui implémente `Iterator`, elle `I32Range` est donc prête à l'emploi :

```
let mut pi = 0.0;
let mut numerator = 1.0;

for k in (I32Range { start: 0, end: 14 }) {
    pi += numerator / (2*k + 1) as f64;
    numerator /= -3.0;
}
pi *= f64::sqrt(12.0);

// IEEE 754 specifies this result exactly.
assert_eq!(pi as f32, std:: f32:: consts::PI);
```

Mais `I32Range` c'est un cas particulier, en ce sens que l'itérable et l'itérateur sont du même type. De nombreux cas ne sont pas si simples. Par exemple, voici le type d'arbre binaire du chapitre 10 :

```
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right:BinaryTree<T>
}
```

La manière classique de parcourir un arbre binaire consiste à effectuer une récurrence, en utilisant la pile d'appels de fonction pour garder une trace de votre place dans l'arbre et des nœuds qui n'ont pas encore été visités. Mais lors de la mise en œuvre `Iterator` pour `BinaryTree<T>`, chaque appel à `next` doit produire exactement une valeur et un retour. Pour garder une trace des nœuds d'arbre qu'il n'a pas encore produits,

l'itérateur doit maintenir sa propre pile. Voici un type d'itérateur possible pour `BinaryTree` :

```
use self:: BinaryTree::*;

// The state of an in-order traversal of a `BinaryTree`.
struct TreeIter<'a, T> {
    // A stack of references to tree nodes. Since we use `Vec`'s
    // `push` and `pop` methods, the top of the stack is the end of the
    // vector.
    //
    // The node the iterator will visit next is at the top of the stack,
    // with those ancestors still unvisited below it. If the stack is empt
    // the iteration is over.
    unvisited:Vec<&'a TreeNode<T>>
}
```

Lorsque nous créons un nouveau `TreeIter`, son état initial devrait être sur le point de produire le nœud feuille le plus à gauche de l'arbre. Selon les règles de la `unvisited` pile, elle devrait donc avoir cette feuille en haut, suivie de ses ancêtres non visités : les nœuds le long du bord gauche de l'arbre. Nous pouvons initialiser `unvisited` en parcourant le bord gauche de l'arbre de la racine à la feuille et en poussant chaque nœud que nous rencontrons, nous allons donc définir une méthode `TreeIter` pour le faire :

```
impl<'a, T: 'a> TreeIter<'a, T> {
    fn push_left_edge(&mut self, mut tree:&'a BinaryTree<T>) {
        while let NonEmpty(ref node) = *tree {
            self.unvisited.push(node);
            tree = &node.left;
        }
    }
}
```

L'écriture `mut tree` permet à la boucle de changer le nœud `tree` vers lequel pointe le long du bord gauche, mais comme il `tree` s'agit d'une référence partagée, elle ne peut pas muter les nœuds eux-mêmes.

Avec cette méthode d'assistance en place, nous pouvons donner `BinaryTree` une `iter` méthode qui renvoie un itérateur sur l'arbre :

```
impl<T> BinaryTree<T> {
    fn iter(&self) -> TreeIter<T> {
        let mut iter = TreeIter { unvisited: Vec::new() };
        iter.push_left_edge(self);
```

```
            iter
        }
    }
```

La `iter` méthode construit a `TreeIter` avec une pile vide `unvisited`, puis appelle `push_left_edge` pour l'initialiser. Le nœud le plus à gauche se retrouve en haut, comme l'exigent les `unvisited` règles de la pile.

Suivant les pratiques de la bibliothèque standard, nous pouvons alors implémenter `IntoIterator` sur une référence partagée à un arbre avec un appel à `BinaryTree::iter` :

```
impl<'a, T: 'a> IntoIterator for &'a BinaryTree<T> {
    type Item = &'a T;
    type IntoIter = TreeIter<'a, T>;
    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}
```

La `IntoIter` définition établit `TreeIter` comme type d'itérateur pour un `&BinaryTree`.

Enfin, dans l' `Iterator` implémentation, nous parcourons réellement l'arbre. Comme `BinaryTree` la `iter` méthode de , la méthode de l'itérateur `next` est guidée par les règles de la pile :

```
impl<'a, T> Iterator for TreeIter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) ->Option<&'a T> {
        // Find the node this iteration must produce,
        // or finish the iteration. (Use the `?` operator
        // to return immediately if it's `None`.)
        let node = self.unvisited.pop()?;

        // After `node`, the next thing we produce must be the leftmost
        // child in `node`'s right subtree, so push the path from here
        // down. Our helper method turns out to be just what we need.
        self.push_left_edge(&node.right);

        // Produce a reference to this node's value.
        Some(&node.element)
    }
}
```

Si la pile est vide, l'itération est terminée. Sinon, `node` est une référence au nœud à visiter maintenant ; cet appel renverra une référence à son `element` champ. Mais d'abord, nous devons avancer l'état de l'itérateur au nœud suivant. Si ce nœud a un sous-arbre droit, le prochain nœud à visiter est le nœud le plus à gauche du sous-arbre, et nous pouvons l'utiliser `push_left_edge` pour le pousser, ainsi que ses ancêtres non visités, sur la pile. Mais si ce nœud n'a pas de sous-arbre droit, `push_left_edge` n'a aucun effet, ce qui est exactement ce que nous voulons : nous pouvons compter sur le nouveau sommet de la pile pour être `node` le premier ancêtre non visité de , le cas échéant.

Avec `IntoIterator` et `Iterator` les implémentations en place, nous pouvons enfin utiliser une `for` boucle pour itérer sur une `BinaryTree` par référence. En utilisant la `add` méthode `BinaryTree` de « Remplir un arbre binaire » :

```
// Build a small tree.
let mut tree = BinaryTree::Empty;
tree.add("jaeger");
tree.add("robot");
tree.add("droid");
tree.add("mecha");

// Iterate over it.
let mut v = Vec::new();
for kind in &tree {
    v.push(*kind);
}
assert_eq!(v, ["droid", "jaeger", "mecha", "robot"]);
```

La figure 15-1 montre comment la `unvisited` pile se comporte lorsque nous parcourons un exemple d'arborescence. À chaque étape, le prochain nœud à visiter est en haut de la pile, avec tous ses ancêtres non visités en dessous.

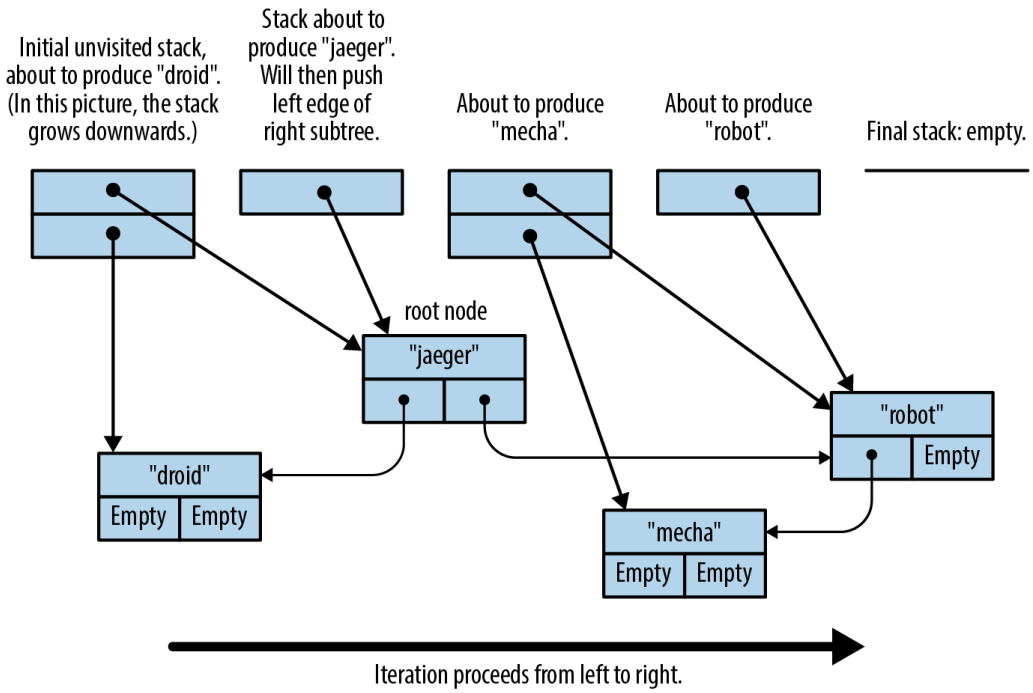

Illustration 15-1. Itérer sur un arbre binaire

Tous les adaptateurs et consommateurs d'itérateurs habituels sont prêts à être utilisés sur nos arbres:

```
assert_eq!(tree.iter()
           .map(|name| format!("mega-{}", name))
           .collect::<Vec<_>>(),
           vec!["mega-droid", "mega-jaeger",
                "mega-mecha", "mega-robot"]);
```

Les itérateurs sont l'incarnation de la philosophie de Rust consistant à fournir des abstractions puissantes et sans coût qui améliorent l'expressivité et la lisibilité du code. Les itérateurs ne remplacent pas entièrement les boucles, mais ils fournissent une primitive capable avec une évaluation paresseuse intégrée et d'excellentes performances.

1 En fait, puisque `Option` est un itérable se comportant comme une séquence de zéro ou un élément, `iterator.filter_map(closure)` est équivalent à `iterator.flat_map(closure)`, en supposant qu'il `closure` renvoie un `Option<T>`.