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 peut 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 à un serveur réseau, des valeurs reçues d'autres threads sur un canal de communication, etc. Et bien sûr, vous pouvez implémenter des itérateurs à vos propres fins. La boucle de Rust fournit une syntaxe naturelle pour l'utilisation des itérateurs, mais les itérateurs eux-mêmes fournissent également un riche ensemble de méthodes pour cartographier, filtrer, joindre, collecter, etc. `for`

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

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

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

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

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

En commençant par le total courant, prend chaque valeur qui produit et applique la fermeture au total courant et à la valeur. La valeur de retour de la fermeture est considérée comme le nouveau total courant. La dernière valeur qu'il renvoie est ce qu'il renvoie lui-même, dans ce cas, le total de la séquence entière. Cela peut sembler étrange si vous êtes habitué et boucles, mais une fois que vous vous y êtes habitué, c'est une alternative lisible et concise. `0 fold 1..=n |sum, item| sum + item fold for while fold`

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

Cet exemple implique une arithmétique simple, mais les itérateurs fonctionnent également bien lorsqu'ils sont utilisés plus lourdement. Ils sont un autre exemple de Rust fournissant des abstractions flexibles qui imposent peu ou pas de frais généraux dans une utilisation typique.

Dans ce chapitre, nous allons vous expliquer :

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

Il y a beaucoup de méthodes, donc c'est bien d'écrémer une section une fois que vous avez l'idée générale. Mais les itérateurs sont très courants dans Rust idiomatique, et être familier avec les outils qui les accompagnent est essentiel pour maîtriser le langage.

Les traits Iterator et IntoIterator

Un itérateur est toute valeur qui implémente le trait
: `std::iter::Iterator`

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

`Item` est le type de valeur produite par l'itérateur. La méthode renvoie , où est la valeur suivante de l'itérateur ou renvoie pour indiquer la fin de la séquence. Ici, nous avons omis les nombreuses méthodes par défaut; nous les couvrirons individuellement tout au long du reste de ce chapitre. `next Some(v) v None Iterator`

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

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

`IntoIter` est le type de la valeur itératrice elle-même, et est le type de valeur qu'il produit. Nous appelons n'importe quel type qui implémente un *itérable*, car c'est quelque chose que vous pourriez itérer si vous le demandiez. `Item IntoIterator`

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

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

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

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

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

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

Avec cet exemple à l'esprit, voici quelques termes pour les itérateurs :

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

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

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

Création d'itérateurs

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

Iter et méthodes iter_mut

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

```
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 : chaque appel à produit une référence à l'élément suivant, jusqu'à ce que nous atteignions la fin du vecteur. `&i32 next`

Chaque type est libre d'être mis en œuvre et de la manière la plus logique pour son objectif. La méthode on renvoie un itérateur qui produit un composant de chemin d'accès à la fois
: `iter iter_mut iter std::path::Path`

```
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 , une tranche empruntée d'une chaîne du type accepté par les appels du système

d'exploitation. `&std::ffi::OsStr`

S'il existe plus d'une façon courante d'itérer sur un type, le type fournit généralement des méthodes spécifiques pour chaque type de traversée, car une méthode simple serait ambiguë. Par exemple, il n'existe aucune méthode sur le type de tranche de chaîne. Au lieu de cela, si est un , renvoie alors un itérateur qui produit chaque octet de , alors qu'il interprète le contenu comme UTF-8 et produit chaque caractère Unicode. `iter iter &str s &str s.bytes() s s.chars()`

Implémentations IntoIterator

Lorsqu'un type implémente , vous pouvez appeler sa méthode vous-même, tout comme une boucle : `IntoIterator into_iter for`

```
// 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 , pour les références partagées (), les références mutables () et les déplacements () : `IntoIterator &T &mut T T`

- Étant donné une *référence partagée* à la collection, renvoie un itérateur qui produit des références partagées à ses éléments. Par exemple, dans le code précédent, renverrait un itérateur dont le type est `.into_iter (&favorites).into_iter() Item &String`
- Étant donné une *référence modifiable* à la collection, renvoie un itérateur qui produit des références modifiables aux éléments. Par exemple, si est certain , l'appel renvoie un itérateur dont le type est `.into_iter vector Vec<String> (&mut vector).into_iter() Item &mut String`
- Lorsque la collection *est passée par valeur*, renvoie un itérateur qui prend possession de la collection et renvoie les éléments par valeur ; la propriété des articles passe de la collection au consommateur, et la col-

lection d'origine est consommée dans le processus. Par exemple, l'appel 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 abandonné, tous les éléments restant dans le sont également abandonnés et l'enveloppe maintenant vide de l'ensemble est éliminée. `into_iter favorites.into_iter() BTreeSet`

Étant donné qu'une boucle s'applique à son opérande, ces trois implémentations sont ce qui crée les idiomes suivants pour itérer sur des références partagées ou modifiables à une collection, ou consommer la collection et s'approprier ses éléments : `for IntoIterator::into_iter`

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

Chacun d'entre eux entraîne simplement un appel à l'une des implémentations répertoriées ici. `IntoIterator`

Tous les types ne fournissent pas les trois implémentations. Par exemple, , et ne pas implémenter sur des références modifiables, car modifier 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, de sorte que la modifier la laisserait mal placée. D'autres types prennent en charge la mutation, mais seulement partiellement. Par exemple, et produire des références modifiables aux valeurs de leurs entrées, mais uniquement des références partagées à leurs clés, pour des raisons similaires à celles données précédemment. `HashSet BTreeSet BinaryHeap IntoIterator HashMap BTreeMap`

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

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

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

`IntoIterator` c'est ce qui fait fonctionner les boucles, donc c'est évidemment nécessaire. Mais lorsque vous n'utilisez pas de boucle, il est plus clair d'écrire que . L'itération par référence partagée est quelque chose dont vous aurez besoin fréquemment, donc et qui est toujours précieux pour leur ergonomie. `for for favorites.iter() (&favorites).into_iter() iter iter_mut`

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

```
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 et , car ce ne sont pas des méthodes d'un trait quelconque: la plupart des types itérables ont juste des méthodes de ces noms. `iter iter_mut`

from_fn et successeurs

Un moyen simple et général de produire une séquence de valeurs consiste à fournir une fermeture qui les renvoie.

Étant donné qu'une fonction renvoie , renvoie un itérateur qui appelle simplement la fonction pour produire ses éléments. Par exemple: `Option<T> std::iter::from_fn`

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

Si chaque élément dépend de celui d'avant, la fonction fonctionne bien. Vous fournissez un élément initial et une fonction qui prend un élément et renvoie un élément du suivant. S'il renvoie , l'itération se termine. Par exemple, voici une autre façon d'écrire la fonction à partir de notre traceur d'ensemble Mandelbrot dans le chapitre 2
: `std::iter::successors Option None escape_time`

```
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'appel produit une séquence de points sur le plan complexe en quadrature répétée du dernier point et en ajoutant le paramètre . Lors du tracé 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 établit une limite sur la durée pendant laquelle nous

poursuivrons la séquence et numérote chaque point, transformant chaque point en un tuple. Nous avons l'habitude de chercher le premier point qui s'éloigne suffisamment de l'origine pour s'échapper. La méthode renvoie un : s'il en existe un ou non. L'appel à se transforme en , mais retourne inchangé : c'est exactement la valeur de retour que nous voulons. `successors c take(limit) enumerate z (i, z) find find Option Some((i, z)) None Option::map Some((i, z)) Some(i) None`

Les deux et acceptent les fermetures, afin que vos fermetures puissent capturer et modifier les variables des étendues environnantes. Par exemple, cette fonction utilise une fermeture pour capturer une variable et l'utiliser comme état d'exécution : `from_fn successors FnMut fibonacci move`

```
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 et 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 l'autre, en passant des fermetures complexes pour obtenir le comportement dont vous avez besoin. Mais cela néglige la possibilité offerte par les itérateurs de clarifier la façon dont les données circulent dans le calcul et d'utiliser des noms standard pour les modèles courants. Assurez-vous de vous être familiarisé avec les autres méthodes d'itération de ce chapitre avant de vous appuyer sur ces deux méthodes; il y a souvent de meilleures façons de faire le travail. `from_fn successors`

méthodes de drainage

De nombreux types de collection fournissent une méthode qui prend une référence modifiable à la collection et renvoie un itérateur qui transmet la propriété de chaque élément au consommateur. Cependant, contrairement à la méthode, qui prend la collection par valeur et la consomme,

emprunte simplement une référence modifiable à la collection, et lorsque l'itérateur est abandonné, il supprime tous les éléments restants de la collection et le laisse vide. `drain into_iter() drain`

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

```
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 devez vider toute la séquence, utilisez la plage complète, , comme argument. `..`

Autres sources d'itérateurs

Les sections précédentes concernent principalement les types de collection tels que les vecteurs et , mais il existe de nombreux autres types dans la bibliothèque standard qui prennent en charge l'itération. Le tableau 15-1 résume les plus intéressants, mais il y en a beaucoup 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). `HashMap`

Tableau 15-1. Autres itérateurs dans la bibliothèque standard

Type ou trait	Expression	Notes
`std::ops::Range`	`1..10`	Les points de terminaison doivent être de type entier pour pouvoir 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. Start doit être un entier. Peut paniquer ou déborder si la valeur atteint la limite du type.
`std::ops::RangeInclusive`	`1..=10`	Comme , mais inclut la valeur finale. `Range`
`Option<T>`	`Some(10).iter()`	Se comporte comme un vecteur dont la longueur est soit 0 () ou 1 (). `None Some(v)`
`Result<T, E>`	`Ok("blah").iter()`	Semblable à , produisant des valeurs. `Option Ok`
`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 non superposées de la longueur donnée, de gauche à droite.
	`v.chunks_mut(1024)`	Comme , mais les tranches sont mutables. `chunks`

Type ou trait	Expression	Notes
	`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 mutables.
	`v.rsplit(...)`	Comme , mais produit des tranches de droite à gauche. `split`
	`v.splitn(n, ...)`	Comme , mais produit au plus des tranches. `split n`
`String`, `&str`	`s.bytes()`	Produit les octets du formulaire UTF-8.
	`s.chars()`	Produit le s que représente l'UTF-8. `char`
	`s.split_whitespace()`	Divise la chaîne par espace et produit des tranches de caractères non spatiaux.
	`s.lines()`	Produit des tranches des lignes de la chaîne.
	`s.split('/')`	Divise la chaîne sur un motif donné, produisant les tranches entre les correspondances. Les motifs peuvent être beaucoup de choses: caractères, chaînes, fermetures.
	`s.matches(char::is_numeric)`	Produit des tranches correspondant au motif donné.

Type ou trait	Expression	Notes
`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 modifiables aux valeurs des entrées.
`std::collections::HashSet`, `std::collections::BTreeSet`	`set1.union(set2)`	Produit des références partagées à des éléments d'union de et . `set1` `set2`
	`set1.intersection(set2)`	Produit des références partagées aux éléments d'intersection de et . `set1` `set2`
`std::sync::mpsc::Receiver`	`recv.iter()`	Produit des valeurs envoyées à partir d'un autre thread sur le fichier . `Sender`
`std::io::Read`	`stream.bytes()`	Produit des octets à partir d'un flux d'E/S.
	`stream.chars()`	Analyse le flux en UTF-8 et produit s. `char`
`std::io::BufRead`	`bufstream.lines()`	Analyse le flux en UTF-8, produit des lignes en s. `String`
	`bufstream.split(0)`	Divise le flux sur un octet donné, produit des tampons inter-octets. `Vec<u8>`

Type ou trait	Expression	Notes
`std::fs::ReadDir`	`std::fs::read_dir(path)`	Produit des entrées d'annuaire.
`std::net::TcpListener`	`listener.incoming()`	Produit des connexions réseau entrantes.
Fonctions libres	`std::iter::empty()`	Renvoie immédiatement. `None`
	`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 trait fournit une large sélection de *méthodes* d'adaptateur, ou simplement des *adaptateurs*, qui consomment un itérateur et en construisent un nouveau avec des comportements utiles. Pour voir comment fonctionnent les adaptateurs, nous allons commencer par deux des adaptateurs les plus populaires, et . Ensuite, nous couvrirons le reste de la boîte à outils de l'adaptateur, couvrant presque toutes les façons que vous pouvez imaginer pour créer des séquences de valeurs à partir d'autres séquences: troncature, saut, combinaison, inversion, concaténation, répétition, etc. `Iterator map filter`

carte et filtre

L'adaptateur du trait vous permet de transformer un itérateur en appliquant une fermeture à ses éléments. L'adaptateur vous permet de filtrer

les éléments d'un itérateur, à l'aide d'une fermeture pour décider lesquels conserver et lesquels déposer. `Iterator map filter`

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

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

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

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

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

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

Un itérateur transmet chaque article à sa fermeture en valeur et, à son tour, transmet la propriété du résultat de la fermeture à son consommateur. Un itérateur transmet chaque article à sa fermeture par référence partagée, en conservant la propriété au cas où l'article serait sélectionné pour être transmis à son consommateur. C'est pourquoi l'exemple doit être déréférencé pour le comparer avec : le type d'élément de l'itérateur est , donc le type de l'argument de fermeture est `.map filter s "iguanas" filter &str s &&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 articles en puisant dans le premier itérateur au besoin. Dans une chaîne d'adaptateurs, la seule façon de faire fonctionner réellement est de faire appel à l'itérateur final. `next`

Ainsi, dans notre exemple précédent, l'appel de méthode lui-même n'analyse en fait aucune ligne de la chaîne; il renvoie simplement un itérateur qui *analyserait* les lignes si on le lui demandait. De même, et il suffit de renvoyer de nouveaux itérateurs qui *mapperaient ou filtreraient* si on le leur demandait. Aucun travail n'a lieu jusqu'à ce qu'il commence à faire appel à l'itérateur. `text.lines() map filter collect 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'appel renvoie un itérateur sur les éléments du tableau et l'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 jamais une valeur de toute la chaîne, donc aucune méthode ne fonctionne jamais. En fait, Rust vous avertira à ce sujet: `iter map next`

```
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. Il est de convention de Rust que les itérateurs doivent faire le minimum de travail nécessaire pour satisfaire chaque appel à ; dans l'exemple, il n'y a pas du tout de tels appels, donc aucun travail n'a lieu. `next`

Le deuxième point important est que les adaptateurs d'itérateur sont une abstraction sans surcharge. Puisque , , et leurs compagnons sont génériques, les appliquer 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 dans son consommateur, puis traduire l'ensemble de l'arrangement en code machine en tant qu'unité. Ainsi, la chaîne // d'itérateurs que nous avons montrée précédemment est aussi efficace que le code que vous écririez probablement à la main: `map filter next lines map filter`

```
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 trait. `Iterator`

filter_map et flat_map

L'adaptateur est correct dans les situations où chaque élément entrant produit un élément sortant. Mais que se passe-t-il si vous souhaitez supprimer certains éléments de l'itération au lieu de les traiter ou remplacer des éléments uniques par zéro ou plusieurs éléments ? Les adaptateurs et vous offrent cette flexibilité. `map filter_map flat_map`

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

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

Par exemple, supposons que vous souhaitiez analyser une chaîne à la recherche de mots séparés par des espaces blancs qui peuvent être analysés en tant que nombres et traiter les nombres en supprimant les autres mots. Vous pouvez 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 les éléments suivants :

```
1.00
0.50
17.00
1.77
```

La fermeture donnée à tente d'analyser chaque tranche séparée par des espaces blancs à l'aide de . Cela renvoie un , qui se transforme en un : une erreur d'analyse devient , alors qu'un résultat d'analyse réussie devient . L'itérateur supprime toutes les valeurs et produit la valeur pour chaque `. filter_map f64::from_str Result<f64, ParseFloatError> .ok() Option<f64> None Some(v) filter_map N one v Some(v)`

Mais quel est l'intérêt de fusionner et de former une seule opération comme celle-ci, au lieu de simplement utiliser ces adaptateurs directement? L'adaptateur affiche 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 de le traiter. Vous pouvez faire la même chose avec seulement et , mais c'est un peu disgracieux: `map filter filter_map filter map`

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

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

La signature de est indiquée ici : `flat_map`

```
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 doit retourner un itérable, mais toute sorte d'itérable fera l'affaire. `flat_map` [1]

Par exemple, supposons que nous ayons un tableau cartographiant les pays à leurs grandes villes. Compte tenu d'une liste de pays, comment pouvons-nous itérer sur leurs grandes 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 les éléments suivants :

```
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 l'imprimons.

Mais rappelez-vous que les itérateurs sont paresseux : ce ne sont que les appels de la boucle à la méthode de l'itérateur qui font que le travail est effectué. 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 puise dans l'itérateur de ville, un élément à la fois, jusqu'à ce qu'il soit épuisé, et seulement ensuite produit un nouvel itérateur de ville pour le pays suivant. L'effet est celui d'une boucle imbriquée, mais emballée pour être utilisée comme itérateur. `for flat_map next`

aplatir

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

```
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
                "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 et ses s de noms sont aplatis dans un itérateur produisant tous les noms. `BTreeMap Vec`

La signature de est la suivante : `flatten`

```
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 être implémentés afin qu'il s'agisse effectivement d'une séquence de séquences. La méthode renvoie ensuite un itérateur sur la concaténation de ces séquences. Bien sûr, cela se fait paresseusement, en dessinant un nouvel élément uniquement lorsque nous avons fini d'itérer sur le dernier. `IntoIterator flatten self`

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

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

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

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

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

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

La raison pour laquelle le est nécessaire est qu'il 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 prend soin de les épisser tous ensemble en quelque chose que nous pouvons enfin dans un . `flatten ch.to_uppercase() flatten collect String`

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

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

prendre et take_while

Les traits et les adaptateurs vous permettent de terminer une itération après un certain nombre d'éléments ou lorsqu'une fermeture décide de

couper les choses. Leurs signatures sont les suivantes
: `Iterator take take_while`

```
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 prennent possession d'un itérateur et renvoient un nouvel itérateur qui transmet les éléments du premier, mettant éventuellement fin à la séquence plus tôt. L'itérateur revient après avoir produit la plupart des articles. L'itérateur s'applique à chaque article et retourne à la place du premier article pour lequel il retourne et à chaque appel ultérieur à
. `take None n take_while predicate None predicate false next`

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

```
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 d'une chaîne se termine par une barre oblique inverse, Rust n'inclut pas l'indentation de la ligne suivante dans la chaîne, de sorte qu'aucune des lignes de la chaîne n'a d'espace blanc de début. Cela signifie que la troisième ligne de est vide. L'adaptateur met fin à l'itération dès qu'il voit cette ligne vide, de sorte que ce code imprime uniquement les deux lignes
: `message take_while`

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

sauter et skip_while

Les traits et les méthodes sont le complément de et : ils laissent tomber un certain nombre d'éléments à partir du début d'une itération, ou laissent tomber des éléments jusqu'à ce qu'une fermeture en trouve un acceptable, puis passent les éléments restants inchangés. Leurs signatures sont les suivantes : `Iterator skip skip_while take take_while`

```
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'adaptateur consiste à ignorer le nom de la commande lors de l'itération sur les arguments de ligne de commande d'un programme. Dans le chapitre 2, notre plus grande calculatrice de dénominateur commun a utilisé le code suivant pour effectuer une boucle sur ses arguments de ligne de commande : `skip`

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

La fonction renvoie un itérateur qui produit les arguments du programme sous la forme 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 de 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. `std::env::args String skip(1)`

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

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

Cela permet de sauter les lignes non vides, mais cet itérateur produit la ligne vide elle-même (après tout, la fermeture renvoyée pour cette ligne. Nous utilisons donc également la méthode pour laisser tomber cela, nous

donnant un itérateur dont le premier élément sera la première ligne du corps du message. Pris avec la déclaration de de la section précédente, ce code imprime: `skip_while false skip message`

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

peekable

Un itérateur jetable vous permet de jeter un coup d'œil à l'article suivant qui sera produit sans le consommer réellement. Vous pouvez transformer n'importe quel itérateur en un itérateur peekable en appelant la méthode du trait : `Iterator peekable`

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

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

Un itérateur a une méthode supplémentaire qui renvoie un : si l'itérateur sous-jacent est fait et sinon , où 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.) `Peekable peek Option<&Item> None Some(r) r`

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

Les itérateurs peekables sont essentiels lorsque vous ne pouvez pas décider du nombre d'articles à consommer à partir d'un itérateur avant d'être 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 fonction permet de 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 revient), nous retournons le nombre que nous avons analysé et laissons le caractère suivant dans l'itérateur, prêt à être consommé. `parse_number peek peek None`

fusible

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

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

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

Itérateurs réversibles et régime

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

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

Vous pouvez penser à un itérateur à double extrémité 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; lorsque les deux se rencontrent, l'itération est terminée :

```
    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 mettre en œuvre : il s'agit littéralement d'une paire de pointeurs vers le début et la fin de la gamme d'éléments que nous n'avons pas encore produits ; et il suffit de dessiner un élément de l'un ou de l'autre. Les itérateurs pour les collections ordonnées aiment et sont également à double extrémité: leur méthode dessine d'abord les plus grands éléments ou entrées. En général, la bibliothèque standard fournit une itération à double extrémité chaque fois que cela est
pratique. `next next_back BTreeSet BTreeMap next_back`

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

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

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

L'itérateur retourné est également à double extrémité: ses méthodes et celles-ci sont simplement échangées: `next 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, et préserver la réversibilité. `map filter`

inspecter

L'adaptateur est 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 fermeture à une référence partagée à chaque élément, puis transmet l'élément. La fermeture ne peut pas affecter les éléments, mais elle peut faire des choses comme les imprimer ou faire des affirmations à leur sujet. `inspect`

Cet exemple montre un cas dans lequel la conversion d'une chaîne en majuscules 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 renvoie un itérateur sur les caractères, pas un seul caractère de remplacement. Le code précédent permet de concaténer toutes les séquences qui retournent en une seule , en imprimant les éléments suivants au fur et à mesure
: `char::to_uppercase flat_map to_uppercase String`

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

chaîne

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

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

```
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'article.

Par exemple:

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

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

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

Un itérateur vérifie si chacun des deux itérateurs sous-jacents est revenu et dirige et appelle l'un ou l'autre, selon le cas. `chain None next next_back`

énumérer

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

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

Il commence par un tampon rectangulaire de pixels:

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

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

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

Et puis il itére sur les bandes, en commençant un fil pour chacune d'elles:

```
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 , où est la tranche du tampon de pixels dans lequel le thread doit dessiner, et est l'index de cette bande dans l'image globale, gracieuseté de l'adaptateur. Compte tenu des limites du tracé et de la taille des bandes, il suffit d'informations pour que le fil détermine quelle partie de l'image lui a été attribuée et donc dans quoi dessiner. `(i, band) band &mut [u8] i enumerate band`

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

fermeture éclair

L'adaptateur combine deux itérateurs en un seul itérateur qui produit des paires contenant une valeur de chaque itérateur, comme une fermeture à glissière reliant ses deux côtés en une seule couture. L'itérateur zippé se termine lorsque l'un des deux itérateurs sous-jacents se termine. `zip`

Par exemple, vous pouvez obtenir le même effet que l'adaptateur en compressant la plage d'extrémité illimitée avec l'autre itérateur : `enumerate 0..`

```
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 penser à une généralisation de : alors qu'attache des indices à la séquence, attache les éléments de tout itérateur arbitraire. Nous avons déjà suggéré que cela puisse aider à fournir un con-

texte pour le traitement des éléments; est une façon plus souple de faire de même. `zip enumerate enumerate zip enumerate zip`

L'argument de n'a pas besoin d'être un itérateur lui-même; il peut être n'importe quel qu'il est possible de le faire : `zip`

```
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 à des itérateurs. Une fois que vous l'avez fait, pouvez-vous jamais retirer l'adaptateur à nouveau? Habituellement, non: les adaptateurs prennent possession de l'itérateur sous-jacent et ne fournissent aucune méthode pour le rendre.

La méthode d'un itérateur emprunte une référence modifiable à l'itérateur afin que vous puissiez appliquer des adaptateurs à la référence. Lorsque vous avez fini de consommer des articles à partir de ces adaptateurs, vous les déposez, l'emprunt se termine et vous retrouvez l'accès à votre itérateur d'origine. `by_ref`

Par exemple, plus tôt dans le chapitre, nous avons montré comment utiliser et 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 , nous pouvons utiliser pour gérer les en-têtes, et lorsque cela est fait, récupérez l'itérateur sous-jacent, qui est parti exactement en position pour gérer le corps du message: `take_while skip_while by_ref take_while take_while`

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

```
Headers:
To: jimb
From: id

Body:
Oooooh, donuts!!
```

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

```
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 est un type d'itérateur, alors est un itérateur aussi, dont et les méthodes s'en remettent à son référent. Lorsque vous appelez un adaptateur sur une référence modifiable à un itérateur, l'adaptateur prend possession de la *référence*, et non de l'itérateur lui-même. C'est

juste un emprunt qui se termine lorsque l'adaptateur sort de sa portée. `I &mut I next size_hint`

cloné, copié

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

```
let a = ['1', '2', '3', '∞'];

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

L'adaptateur est la même idée, mais plus restrictive : le type de référent doit implémenter . Un appel comme est à peu près le même que . Étant donné que chaque type qui implémente également , est strictement plus général, mais en fonction du type d'élément, un appel peut effectuer des quantités arbitraires d'allocation et de copie. Si vous supposez que cela ne se produirait jamais parce que votre type d'article est quelque chose de simple, il est préférable d'utiliser pour que le vérificateur de type vérifie vos hypothèses. `copied Copy iter.copied() iter.map(|r| *r) Copy Clone cloned clone copied`

cycle

L'adaptateur renvoie un itérateur qui répète à l'infini la séquence produite par l'itérateur sous-jacent. L'itérateur sous-jacent doit être implémenté afin de pouvoir enregistrer son état initial et le réutiliser chaque fois que le cycle redémarre. `cycle std::clone::Clone 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 une 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 , et tout nombre divisible par cinq avec . Les nombres divisibles par les deux deviennent . `fizz buzz fizzbuzz`

Itérateurs consommateurs

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

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

Accumulation simple: nombre, somme, produit

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

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

fn main() {
```

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

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

```
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 et travailler avec d'autres types en implémentant les et traits, que nous ne décrirons pas dans ce livre.) `sum product std::iter::Sum std::iter::Product`

max, min

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

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

Comme expliqué dans « Comparaisons d'équivalence », les types à virgule flottante de Rust et implémentent uniquement , pas , de sorte que vous ne pouvez pas utiliser les méthodes et 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 de telles fonctions devraient faire avec les valeurs IEEE NaN. Le simple fait de les ignorer risquerait de masquer des problèmes plus

graves dans le
code. `f32 f64 std::cmp::PartialOrd std::cmp::Ord min max`

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

max_by, min_by

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

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

max_by_key, min_by_key

Les méthodes et sur vous permettent 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, et pas seulement à l'extremum lui-même, ces fonctions sont souvent plus utiles que et . Leurs signatures sont les suivantes
: `max_by_key min_by_key Iterator min max`

```
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 article et retourne tout type commandé, retournez l'article pour lequel la fermeture a retourné le maximum ou le minimum , ou si aucun article n'a été produit. `B B None`

Par exemple, si vous devez scanner une table de hachage des villes pour trouver les villes avec les plus grandes et les plus petites populations, 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 est appliquée à chaque élément produit par l'itérateur et renvoie la valeur à utiliser à des fins de comparaison, dans ce cas, la population de la ville. La valeur renvoyée est l'article 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 le tableau.) `|&(_name, pop)| pop`

Comparaison des séquences d'éléments

Vous pouvez utiliser les opérateurs et pour comparer des chaînes, des vecteurs et des 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 telles

que et qui font le même travail, en tirant des paires d'éléments des itérateurs et en les comparant jusqu'à ce qu'une décision puisse être prise. Par exemple: `< == eq lt`

```
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()));
```

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

Les itérateurs fournissent les méthodes et les méthodes pour les comparaisons d'égalité, et , , , et les méthodes pour les comparaisons ordonnées. Les méthodes et se comportent comme les méthodes correspondantes des et traits. `eq ne lt le gt ge cmp partial_cmp Ord PartialOrd`

tout et n'importe quoi

Les méthodes et appliquent une fermeture à chaque article produit par l'itérateur et retournent si la fermeture revient pour un article ou pour tous les articles : `any all true true`

```
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 pour un article donné, elle revient immédiatement, sans tirer d'autres éléments de l'itérateur. `true any true`

position, rposition et ExactSizeIterator

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

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

La méthode est la même, sauf qu'elle recherche à partir de la droite. Par exemple : `rposition`

```
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 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 attribuer des indices de la même manière, en commençant par à gauche. Un itérateur de taille exacte est celui qui implémente le
trait: `rposition position 0 std::iter::ExactSizeIterator`

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

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

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

plier et rfold

La méthode est un outil très général pour accumuler une sorte de résultat sur toute la séquence d'éléments produits par un itérateur. Compte tenu d'une valeur initiale, que nous appellerons *l'accumulateur*, et d'une fermeture, 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 considérée comme le nouvel accumulateur, à transmettre à la fermeture avec l'élément suivant. La valeur finale de l'accumulateur est ce qui lui-même retourne. Si la séquence est vide, renvoie simplement l'accumulateur initial. `fold fold fold fold`

Beaucoup d'autres méthodes pour consommer les 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 signature de la méthode est la suivante : `fold`

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

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

Notez que les valeurs de l'accumulateur sont déplacées dans et hors de la fermeture, de sorte que vous pouvez utiliser avec des types non-accumulateur : `fold Copy`

```
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 méthode est identique à , sauf qu'elle nécessite un itérateur à double extrémité et traite ses éléments du dernier au premier : `rfold fold`

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

Votre fermeture peut renvoyer l'un des types suivants, indiquant comment le pliage doit se dérouler:

- Si votre fermeture revient, peut-être parce qu'elle effectue des E/S ou effectue une autre opération faillible, alors le retour indique de continuer le pliage, avec comme nouvelle valeur d'accumulateur. Le retour provoque l'arrêt immédiat du pliage. La valeur finale du pli est une valeur portante sur l'accumulateur final, ou l'erreur renvoyée par la fermeture. `Result<T, E> Ok(v) try_fold v Err(e) Result`
- Si votre fermeture revient , indique alors que le pliage doit continuer avec comme nouvelle valeur d'accumulateur et indique que l'itération doit s'arrêter immédiatement. La valeur finale du pli est également un fichier . `Option<T> Some(v) v None Option`
- Enfin, la fermeture peut renvoyer une valeur. Ce type est un enum avec deux variantes, et , ce qui signifie continuer avec une nouvelle valeur d'accumulateur , ou s'arrêter tôt. Le résultat du pli est une valeur : si le pli a consommé l'itérateur entier, donnant la valeur finale de l'accumulateur ; ou , si la fermeture a renvoyé cette valeur. `std::ops::ControlFlow Continue(c) Break(b) c ControlFlow Continue(v) v Break(b)` `Continue(c)` et se comporter exactement comme et . L'avantage d'utiliser au lieu de 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 tôt. Nous en montrons un exemple ci-dessous. `Break(b) Ok(c) Err(b) ControlFlow Result`

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

Parce qu'il est si flexible, il est utilisé pour mettre en œuvre de nombreuses autres méthodes de consommation. Par exemple, voici une implémentation de : `try_fold Iterator 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 : promet d'arrêter de consommer des éléments de l'itérateur sous-jacent dès qu'il renvoie false, mais consomme toujours l'itérateur entier. `fold all predicate fold`

Si vous implémentez votre propre type d'itérateur, il vaut la peine d'examiner si votre itérateur pourrait implémenter plus efficacement que la

définition par défaut du trait. Si vous pouvez accélérer, toutes les autres méthodes construites dessus en bénéficieront également. `try_fold Iterator try_fold`

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

nième, nth_back

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

Il ne prend pas possession de 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 présentée ici :

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

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

dernier

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

```
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 l'avant, 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 devez simplement écrire . `iter.next_back()`

find, rfind et find_map

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

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

La méthode est similaire, mais elle nécessite un itérateur à double extrémité et recherche les valeurs de l'arrière vers l'avant, en renvoyant le *dernier* élément pour lequel la fermeture renvoie . `rfind true`

Par exemple, en utilisant le tableau des villes et des populations de « max_by_key, min_by_key », vous pourriez 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 seulement 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, c'est exactement ce que vous voulez. Sa signature est : `find_map`

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

C'est comme , sauf qu'au lieu de retourner , la fermeture devrait renvoyer une certaine valeur. renvoie le premier qui est

`.find bool Option find_map Option Some`

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

```
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")));
```

Building Collections: collect et FromIterator

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

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

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

```
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, elle-même ne sait pas comment construire tous ces types. Au contraire, lorsqu'un type de collection aime ou sait comment se construire à partir d'un itérateur, il implémente le trait, pour lequel il ne s'agit que d'un placage
pratique: `collect Vec HashMap std::iter::FromIterator collect`

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

Si un type de collection implémente , sa fonction associée au type génère une valeur de ce type à partir d'un élément de production itérable de type . `FromIterator<A> from_iter 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, la mise en œuvre de fonctionne de cette
façon. `std::collections::LinkedList FromIterator`

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

```
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 sa mémoire tampon, nécessitant un appel à l'allocateur de tas et une copie des éléments existants. Les vecteurs prennent des mesures algorithmiques pour maintenir cette surcharge basse, mais s'il y avait un moyen d'allouer simplement un tampon de la bonne taille pour commencer, il n'y aurait pas besoin de redimensionner du tout.

C'est là qu'intervient la méthode du trait : `Iterator 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 un , par exemple, sait exactement combien d'éléments il produira, tout comme un itérateur sur un ou . Ces itérateurs fournissent leurs propres définitions spécialisées pour . `Range Vec HashMap size_hint`

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

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

```
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 argument de , l'inférence de type de Rust vous le fournira. Lorsque vous épelez le type de , vous vous assurez que c'est le cas. `collect args`

Le trait d'extension

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

```
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 , elles ont donc toutes cette méthode; il en va de même pour . Les tableaux et les tranches, qui ont une longueur fixe, ne le font pas. `Extend String`

La définition du trait est la suivante :

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

Évidemment, c'est très similaire à : cela crée une nouvelle collection, alors qu'elle étend une collection existante. En fait, plusieurs implémentations de dans la bibliothèque standard créent simplement une nouvelle collection vide, puis appellent pour la remplir. Par exemple, l'implémentation de for fonctionne de cette façon: `std::iter::FromIterator Extend FromIterator extend FromIterator std::collections::LinkedList`

```
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
    }
}
```

partition

La méthode divise les éléments d'un itérateur entre deux collections, en utilisant une fermeture pour décider de l'appartenance de chaque élément : `partition`

```
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 , peut faire n'importe quel type de collections que vous aimez, bien que les deux doivent être du même type. Et comme , vous devrez spécifier le type de retour : l'exemple précédent écrit le type de et permet à l'inférence de type de choisir les bons paramètres de type pour l'appel à `.collect partition collect living nonliving partition`

La signature de est la suivante : `partition`

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

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

D'autres langages proposent des 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 construire une collection d'une sorte ou d'une autre à l'interne, de toute façon. `partition`

for_each et try_for_each

La méthode applique simplement une fermeture à chaque élément: `for_each`

```
["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);
        });
```

Cette impression est :

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

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

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

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)
    })?;
```

Mise en œuvre de vos propres itérateurs

Vous pouvez implémenter les caractéristiques et pour vos propres types, ce qui rend tous les adaptateurs et consommateurs présentés dans ce chapitre disponibles pour utilisation, ainsi que de nombreux autres codes de bibliothèque et de caisse 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. `IntoIterator Iterator`

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

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

L'itération sur un 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 type lui-même, en utilisant comme valeur suivante et comme limite. Vous pouvez donc mettre en œuvre de la sorte : `I32Range I32Range start end Iterator`

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

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

```
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 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 : `I32Range`

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

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

La façon classique de marcher dans un arbre binaire est de se récurser, en utilisant la pile d'appels de fonction pour garder une trace de votre place dans l'arbre et des nœuds à visiter. Mais lors de la mise en œuvre pour , chaque appel à doit produire exactement une valeur et un retour. Pour garder une trace des nœuds d'arborescence qu'il n'a pas encore produits, l'itérateur doit maintenir sa propre pile. Voici un type d'itérateur possible pour : `Iterator BinaryTree<T> next 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 e
        // the iteration is over.
        unvisited: Vec<&'a TreeNode<T>>
    }
```

Lorsque nous créons un nouveau , son état initial devrait être sur le point de produire le nœud foliaire le plus à gauche de l'arbre. Selon les règles de la pile, elle devrait donc avoir cette feuille sur le dessus, suivie de ses ancêtres non visités: les nœuds le long du bord gauche de l'arbre. Nous pouvons initialiser 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 pour le faire: `TreeIter unvisited unvisited TreeIter`

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

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

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

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

Suivant les pratiques de la bibliothèque standard, on peut ensuite implémenter sur une référence partagée à un arbre avec un appel à

: IntoIterator 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 définition établit comme type d'itérateur pour un fichier
. IntoIter TreeIter &BinaryTree

Enfin, dans la mise en œuvre, nous pouvons réellement marcher dans l'arbre. Comme la méthode de , la méthode de l'itérateur est guidée par les règles de la pile : Iterator BinaryTree iter next

```
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, est une référence au nœud à visiter maintenant; cet appel renverra une référence à son champ. Mais d'abord, nous devons faire progresser l'état de l'itérateur vers le nœud suivant. Si ce nœud a un sous-arbre droit, le nœud suivant à visiter est le nœud le plus à gauche du sous-arbre, et nous pouvons l'utiliser pour le pousser, ainsi que ses ancêtres non visités, sur la pile. Mais si ce nœud n'a pas de sous-arbre droit, n'a aucun effet, ce qui est exactement ce que nous voulons: nous pouvons compter sur le nouveau

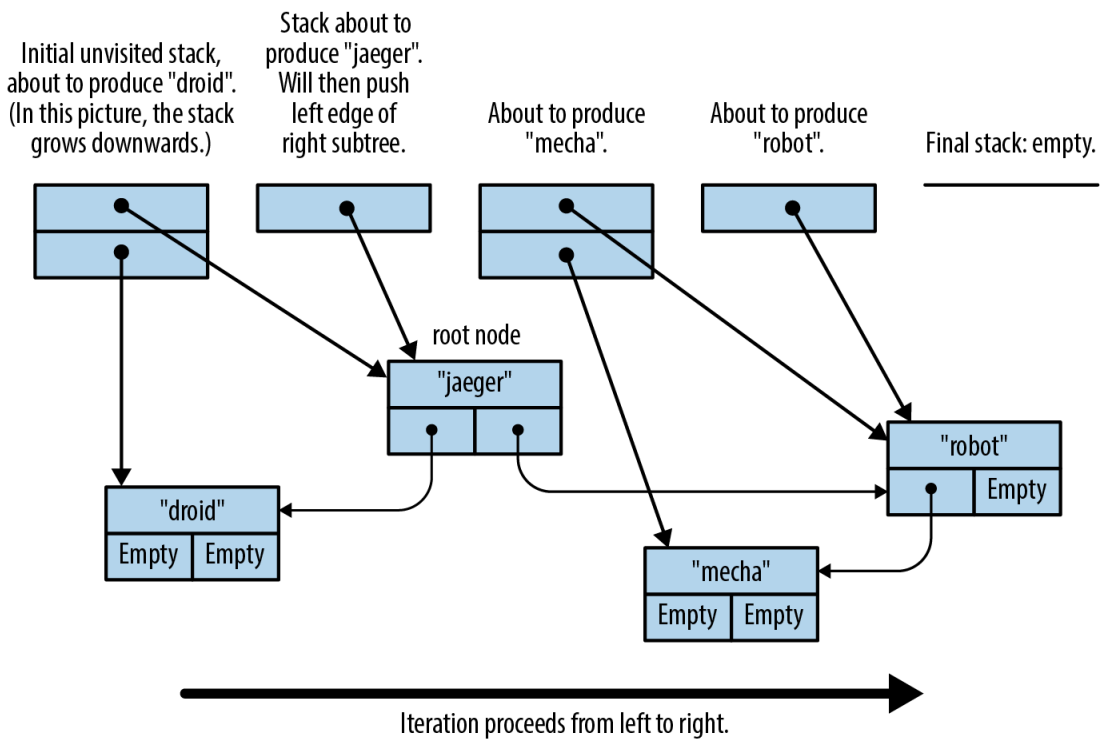
sommet de la pile pour être le premier ancêtre non visité, le cas échéant. `node element push_left_edge push_left_edge node`

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

```
// 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 pile se comporte lorsque nous itérons dans un arbre d'échantillonnage. À chaque étape, le prochain nœud à visiter se trouve en haut de la pile, avec tous ses ancêtres non visités en dessous. `unvisited`



Graphique 15-1. Itération sur un arbre binaire

Tous les adaptateurs d'itérateurs habituels et les consommateurs 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 de 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, puisqu'est un itérable se comportant comme une suite de zéro ou un élément, est équivalent à , en supposant renvoie un `.Option iterator.filter_map (closure) iterator.flat_map(closure) closure Option<T>`