Chapitre 16. Collections

> *Nous nous comportons tous comme le démon de Maxwell. Les organismes s'organisent. Dans l'expérience quotidienne réside la raison pour laquelle les physiciens sobres à travers deux siècles ont maintenu ce fantasme de dessin animé vivant. Nous trions le courrier, construisons des châteaux de sable, résolvons des puzzles, séparons le bon grain de l'ivraie, réarrangeons les pièces d'échecs, collectionnons des timbres, alphabétisons les livres, créons une symétrie, composons des sonnets et des sonates, et mettons de l'ordre dans nos pièces, et tout cela nous ne nécessite pas une grande énergie, tant que nous pouvons appliquer l'intelligence.*
>
> —James Gleick, *L'information : une histoire, une théorie, un déluge*

La bibliothèque standard Rust contient plusieurs *collections*, des types génériques pour stocker des données en mémoire. Nous avons déjà utilisé des collections, telles que et , tout au long de ce livre. Dans ce chapitre, nous couvrirons en détail les méthodes de ces deux types, ainsi que les autres demi-douzaines de collections standard. Avant de commencer, abordons quelques différences systématiques entre les collections de Rust et celles d'autres langues. `Vec` `HashMap`

Tout d'abord, les déménagements et les emprunts sont partout. Rust utilise des mouvements pour éviter la copie en profondeur des valeurs. C'est pourquoi la méthode prend son argument par valeur, pas par référence. La valeur est déplacée dans le vecteur. Les diagrammes du chapitre 4 montrent comment cela fonctionne dans la pratique : pousser un Rust vers a est rapide, car Rust n'a pas besoin de copier les données de caractère de la chaîne, et la propriété de la chaîne est toujours claire. `Vec<T>::push(item)` `String` `Vec<String>`

Deuxièmement, Rust n'a pas d'erreurs d'invalidation, le genre de bogue de pointeur pendant où une collection est redimensionnée ou modifiée d'une autre manière, alors que le programme contient un pointeur vers les données qu'elle contient. Les erreurs d'invalidation sont une autre source de comportement indéfini en C++, et elles provoquent occasionnellement même dans les langages sécurisés pour la mémoire. Le vérificateur d'emprunt de Rust les exclut au moment de la compilation. `ConcurrentModificationException`

Enfin, Rust n'a pas , nous verrons donc s dans des endroits où d'autres langues utiliseraient . `null Option null`

En dehors de ces différences, les collections de Rust sont à peu près ce à quoi vous vous attendez. Si vous êtes un programmeur expérimenté pressé, vous pouvez parcourir ici, mais ne manquez pas « Entrées ».

Aperçu

Le tableau 16-1 montre les huit collections standard de Rust. Tous sont des types génériques.

Tableau 16-1. Résumé des collections standard

Collection	Description	Type de collection similaire dans…		
		C++	Java	Python
`Vec<T>`	Baie extensible	`vector`	`ArrayList`	`list`
`VecDeque<T>`	File d'attente à double extrémité (tampon annulaire extensible)	`deque`	`ArrayDeque`	`collections.deque`
`LinkedList<T>`	Liste doublement liée	`list`	`LinkedList`	—
`BinaryHeap<T>` `where T: Ord`	Nombre maximal de tas	`priority_queue`	`PriorityQueue`	`heapq`
`HashMap<K, V>` `where K: Eq + Hash`	Table de hachage clé-valeur	`unordered_map`	`HashMap`	`dict`
`BTreeMap<K, V>` `where K: Ord`	Table clé-valeur triée	`map`	`TreeMap`	—
`HashSet<T>` `where T: Eq + Hash`	Ensemble non ordonné basé sur le hachage	`unordered_set`	`HashSet`	`set`

Collection	Description	Type de collection similaire dans…		
		C++	Java	Python
`BTreeSet<T>` `where T: Ord`	Ensemble trié	`set`	`TreeSet`	—

`Vec<T>`, et sont les types de collection les plus généralement utiles. Les autres ont des utilisations de niche. Ce chapitre traite de chaque type de collection à tour de rôle : `HashMap<K, V>` `HashSet<T>`

`Vec<T>`

Tableau de valeurs de type . Environ la moitié de ce chapitre est consacrée à ses nombreuses méthodes utiles. `T` `Vec`

`VecDeque<T>`

Comme , mais mieux pour une utilisation en tant que file d'attente premier entré, premier sorti. Il prend en charge l'ajout et la suppression efficaces de valeurs au début de la liste ainsi qu'à l'arrière. Cela se fait au prix d'un léger ralentissement de toutes les autres opérations. `Vec<T>`

`BinaryHeap<T>`

Une file d'attente prioritaire. Les valeurs de a sont organisées de manière à ce qu'il soit toujours efficace de rechercher et de supprimer la valeur maximale. `BinaryHeap`

`HashMap<K, V>`

Table de paires clé-valeur. La recherche d'une valeur par sa clé est rapide. Les entrées sont stockées dans un ordre arbitraire.

`BTreeMap<K, V>`

Comme , mais il conserve les entrées triées par clé. A stocke ses entrées dans l'ordre de comparaison. À moins que vous n'ayez besoin que les entrées restent triées, a est plus rapide. `HashMap<K, V>` `BTreeMap<String, i32>` `String` `HashMap`

`HashSet<T>`

Ensemble de valeurs de type . L'ajout et la suppression de valeurs sont rapides, et il est rapide de demander si une valeur donnée est dans l'ensemble ou non. `T`

`BTreeSet<T>`

Comme , mais il conserve les éléments triés par valeur. Encore une fois, à moins que vous n'ayez besoin que les données soient triées, a est plus rapide. `HashSet<T> HashSet`

Parce qu'il est rarement utilisé (et qu'il existe de meilleures alternatives, à la fois en termes de performances et d'interface, pour la plupart des cas d'utilisation), nous ne le décrivons pas ici. `LinkedList`

Vec<T>

Nous supposerons une certaine familiarité avec , puisque nous l'avons utilisé tout au long du livre. Pour une introduction, voir « Vecteurs ». Ici, nous allons enfin décrire en profondeur ses méthodes et son fonctionnement interne. `Vec`

Le moyen le plus simple de créer un vecteur est d'utiliser la macro : `vec!`

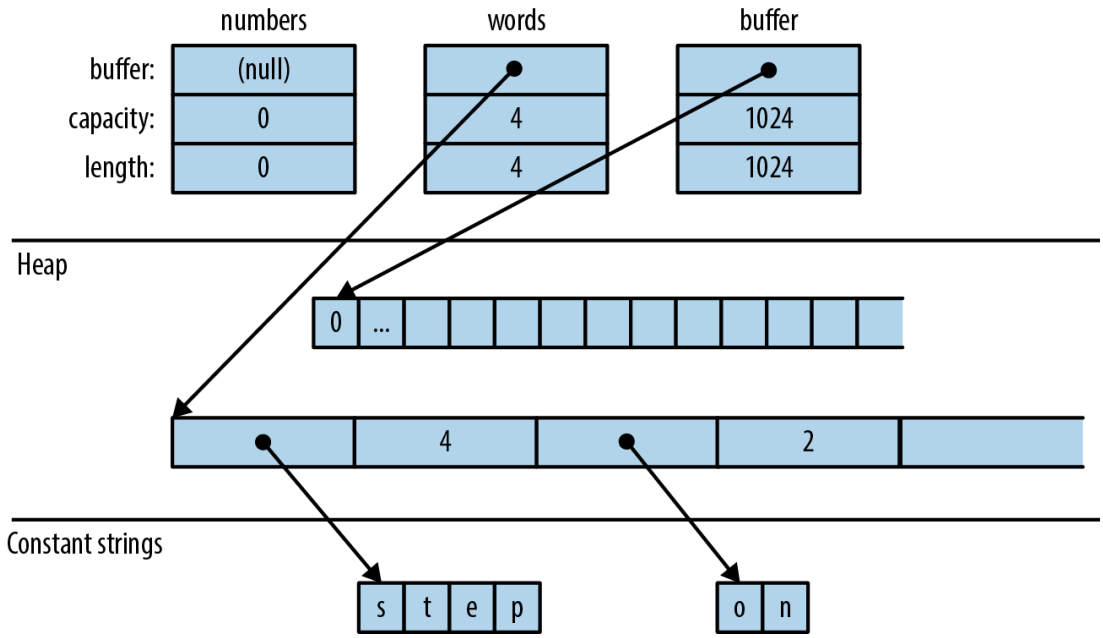
```
// Create an empty vector
let mut numbers: Vec<i32> = vec![];

// Create a vector with given contents
let words = vec!["step", "on", "no", "pets"];
let mut buffer = vec![0u8; 1024];  // 1024 zeroed-out bytes
```

Comme décrit au chapitre 4, un vecteur comporte trois champs : la longueur, la capacité et un pointeur vers une allocation de tas où les éléments sont stockés. La figure 16-1 montre comment les vecteurs précédents apparaîtraient en mémoire. Le vecteur vide, , a initialement une capacité de 0. Aucune mémoire de tas n'est allouée tant que le premier élément n'est pas ajouté. `numbers`

Comme toutes les collections, implémente , de sorte que vous pouvez créer un vecteur à partir de n'importe quel itérateur en utilisant la méthode de l'itérateur, comme décrit dans « Building Collections: collect and FromIterator »: `Vec std::iter::FromIterator .collect()`

```
// Convert another collection to a vector.
let my_vec = my_set.into_iter().collect::<Vec<String>>();
```



Graphique 16-1. Disposition vectorielle en mémoire : chaque élément de mots est une valeur composée d'un pointeur et d'une longueur `&str`

Accès aux éléments

L'obtention d'éléments d'un tableau, d'une tranche ou d'un vecteur par index est simple :

```
// Get a reference to an element
let first_line = &lines[0];

// Get a copy of an element
let fifth_number = numbers[4];        // requires Copy
let second_line = lines[1].clone();   // requires Clone

// Get a reference to a slice
let my_ref = &buffer[4..12];

// Get a copy of a slice
let my_copy = buffer[4..12].to_vec();  // requires Clone
```

Toutes ces formes paniquent si un index est hors limites.

Rust est pointilleux sur les types numériques, et il ne fait aucune exception pour les vecteurs. Les longueurs de vecteurs et les indices sont de type . Essayer d'utiliser un , , ou comme index vectoriel est une erreur. Vous pouvez utiliser un cast pour convertir si nécessaire; voir « Type Casts ». `usize u32 u64 isize n as usize`

Plusieurs méthodes permettent d'accéder facilement à des éléments particuliers d'un vecteur ou d'une tranche (notez que toutes les méthodes de tranche sont également disponibles sur les tableaux et les vecteurs) :

slice.first()

Renvoie une référence au premier élément de , le cas échéant. `slice`

Le type de retour est , donc la valeur de retour est si est vide et si elle n'est pas vide : `Option<&T> None slice Some(&slice[0])`

```
if let Some(item) = v.first() {
    println!("We got one! {}", item);
}
```

slice.last()

Similaire mais renvoie une référence au dernier élément.

slice.get(index)

Renvoie la référence à , s'il existe. Si a moins d'éléments, cela renvoie : `Some slice[index] slice index+1 None`

```
let slice = [0, 1, 2, 3];
assert_eq!(slice.get(2), Some(&2));
assert_eq!(slice.get(4), None);
```

slice.first_mut(),, slice.last_mut() slice.get_mut(index)

Variations de ce qui précède qui empruntent des références: `mut`

```
let mut slice = [0, 1, 2, 3];
{
    let last = slice.last_mut().unwrap();   // type of last: &mut i
    assert_eq!(*last, 3);
    *last = 100;
}
assert_eq!(slice, [0, 1, 2, 100]);
```

Étant donné que le renvoi d'une valeur by signifierait son déplacement, les méthodes qui accèdent aux éléments en place renvoient généralement ces éléments par référence. `T`

Une exception est la méthode, qui fait des copies: `.to_vec()`

slice.to_vec()

Clone une tranche entière, renvoyant un nouveau vecteur :

```
let v = [1, 2, 3, 4, 5, 6, 7, 8, 9];
assert_eq!(v.to_vec(),
```

```
                vec![1, 2, 3, 4, 5, 6, 7, 8, 9]);
    assert_eq!(v[0..6].to_vec(),
                vec![1, 2, 3, 4, 5, 6]);
```

Cette méthode n'est disponible que si les éléments sont clonables, c'est-à-dire . `where T: Clone`

Itération

Les vecteurs, les tableaux et les tranches sont itérables, soit par valeur, soit par référence, selon le modèle décrit dans « Implémentations IntoIterator » :

- L'itération sur un ou un tableau produit des éléments de type . Les éléments sont déplacés hors du vecteur ou du tableau un par un, le consommant. `Vec<T>` `[T; N]` `T`
- L'itération sur une valeur de type , , ou, c'est-à-dire une référence à un tableau, une tranche ou un vecteur, produit des éléments de type , des références aux éléments individuels, qui ne sont pas déplacés. `&[T; N]` `&[T]` `&Vec<T>` `&T`
- Itération sur une valeur de type , , ou produit des éléments de type . `&mut [T; N]` `&mut [T]` `&mut Vec<T>` `&mut T`

Les tableaux, les tranches et les vecteurs ont également des méthodes (décrites dans « iter and iter_mut Methods ») pour créer des itérateurs qui produisent des références à leurs éléments. `.iter()` `.iter_mut()`

Nous couvrirons quelques façons plus sophistiquées d'itérer sur une tranche dans « Splitting ».

Vecteurs de croissance et de rétrécissement

La *longueur* d'un tableau, d'une tranche ou d'un vecteur est le nombre d'éléments qu'il contient :

`slice.len()`

Renvoie la longueur d'un . `slice` `usize`

`slice.is_empty()`

Est vrai si ne contient aucun élément (c'est-à-dire). `slice` `slice.len() == 0`

Les autres méthodes de cette section concernent la croissance et le rétrécissement des vecteurs. Ils ne sont pas présents sur les tableaux et les

tranches, qui ne peuvent pas être redimensionnés une fois créés.

Tous les éléments d'un vecteur sont stockés dans un morceau de mémoire contigu et alloué au tas. La *capacité* d'un vecteur est le nombre maximal d'éléments qui pourraient tenir dans ce morceau. gère normalement la capacité pour vous, allouant automatiquement un tampon plus grand et y déplaçant les éléments lorsque plus d'espace est nécessaire. Il existe également quelques méthodes pour gérer explicitement la capacité : `Vec`

`Vec::with_capacity(n)`

Crée un nouveau vecteur vide avec capacité . `n`

`vec.capacity()`

Renvoie la capacité de , sous la forme d'un fichier . Il est toujours vrai que . `vec usize vec.capacity() >= vec.len()`

`vec.reserve(n)`

S'assure que le vecteur a au moins une capacité de réserve suffisante pour plus d'éléments: c'est-à-dire est au moins . S'il y a déjà assez de place, cela ne fait rien. Si ce n'est pas le cas, cela alloue un tampon plus grand et y déplace le contenu du vecteur. `n vec.capacity() vec.len() + n`

`vec.reserve_exact(n)`

Comme , mais dit de ne pas allouer de capacité supplémentaire pour la croissance future, au-delà de . Après, c'est exactement . `vec.reserve(n) vec n vec.capacity() vec.len() + n`

`vec.shrink_to_fit()`

Essaie de libérer la mémoire supplémentaire si elle est supérieure à . `vec.capacity() vec.len()`

`Vec<T>` a de nombreuses méthodes qui ajoutent ou suppriment des éléments, modifiant la longueur du vecteur. Chacun d'entre eux prend son argument par référence. `self mut`

Ces deux méthodes ajoutent ou suppriment une seule valeur à la fin d'un vecteur :

`vec.push(value)`

Ajoute le donné à la fin de . `value vec`

`vec.pop()`

Supprime et renvoie le dernier élément. Le type de retour est . Cela renvoie si l'élément éclaté est et si le vecteur était déjà vide. `Option<T> Some(x) x None`

Notez que prend son argument par valeur, pas par référence. De même, renvoie la valeur poppée, pas une référence. Il en va de même pour la plupart des méthodes restantes de cette section. Ils déplacent les valeurs à l'intérieur et à l'extérieur des vecteurs. `.push()` `.pop()`

Ces deux méthodes ajoutent ou suppriment une valeur n'importe où dans un vecteur :

`vec.insert(index, value)`

Insère le donné à , en faisant glisser toutes les valeurs existantes à un endroit vers la droite pour faire de la place. `value` `vec[index]` `vec[index..]`

Panique si . `index > vec.len()`

`vec.remove(index)`

Supprime et renvoie , en faisant glisser toutes les valeurs existantes à un endroit vers la gauche pour combler l'écart. `vec[index]` `vec[index+1..]`

Panique si , puisque dans ce cas il n'y a pas d'élément à supprimer. `index >= vec.len()` `vec[index]`

Plus le vecteur est long, plus cette opération est lente. Si vous vous retrouvez à faire beaucoup, envisagez d'utiliser un (expliqué dans « VecDeque<T> ») au lieu d'un . `vec.remove(0)` `VecDeque` `Vec`

Les deux et sont plus lents plus les éléments doivent être déplacés. `.insert()` `.remove()`

Quatre méthodes modifient la longueur d'un vecteur en une valeur spécifique :

`vec.resize(new_len, value)`

Définit la longueur de . Si cela augmente la longueur, des copies de sont ajoutées pour remplir le nouvel espace. Le type d'élément doit implémenter le trait. `vec` `new_len` `vec` `value` `Clone`

`vec.resize_with(new_len, closure)`

Tout comme , mais appelle la fermeture pour construire chaque nouvel élément. Il peut être utilisé avec des vecteurs d'éléments qui ne sont pas . `vec.resize` `Clone`

`vec.truncate(new_len)`

Réduit la longueur de à , en laissant tomber tous les éléments qui se trouvaient dans la plage . `vec new_len vec[new_len..]`

Si est déjà inférieur ou égal à , rien ne se passe. `vec.len() new_len`

`vec.clear()`

Supprime tous les éléments de . C'est la même chose que . `vec vec.truncate(0)`

Quatre méthodes ajoutent ou suppriment plusieurs valeurs à la fois :

`vec.extend(iterable)`

Ajoute tous les éléments de la valeur donnée à la fin de , dans l'ordre. C'est comme une version à valeurs multiples de . L'argument peut être n'importe quoi qui implémente . `iterable vec .push() iterable IntoIterator<Item=T>`

Cette méthode est si utile qu'il existe un trait standard, le trait, que toutes les collections standard implémentent. Malheureusement, cela provoque un regroupement avec d'autres méthodes de trait dans une grande pile au bas du code HTML généré, il est donc difficile de trouver quand vous en avez besoin. Vous n'avez qu'à vous rappeler qu'il est là! Voir « The Extend Trait » pour plus d'informations. `Extend rustdoc .extend()`

`vec.split_off(index)`

Comme , sauf qu'il renvoie un contenant les valeurs supprimées de la fin de . C'est comme une version à valeurs multiples de . `vec.truncate(index) Vec<T> vec .pop()`

`vec.append(&mut vec2)`

Cela déplace tous les éléments de dans , où est un autre vecteur de type . Après, est vide. `vec2 vec vec2 Vec<T> vec2`

C'est comme si cela existait encore par la suite, avec sa capacité non affectée. `vec.extend(vec2) vec2`

`vec.drain(range)`

Cela supprime le de et renvoie un itérateur sur les éléments supprimés, où est une valeur de plage, comme ou . `range vec[range] vec range .. 0..4`

Il existe également quelques méthodes bizarres pour supprimer sélectivement certains éléments d'un vecteur :

`vec.retain(test)`

Supprime tous les éléments qui ne réussissent pas le test donné. L'argument est une fonction ou une fermeture qui implémente . Pour chaque élément de , cet appel , et s'il retourne , l'élément est supprimé du vecteur et supprimé. `test FnMut(&T) -> bool vec test(&element) false`

En dehors de la performance, c'est comme écrire :

```
vec = vec.into_iter().filter(test).collect();
```

`vec.dedup()`

Laisse tomber les éléments répétés. C'est comme l'utilitaire shell Unix. Il recherche les endroits où les éléments adjacents sont égaux et supprime les valeurs égales supplémentaires afin qu'il n'en reste qu'une seule : `uniq vec`

```
let mut byte_vec = b"Mississsssissippi".to_vec();
byte_vec.dedup();
assert_eq!(&byte_vec, b"Misisipi");
```

Notez qu'il y a encore deux caractères dans la sortie. Cette méthode supprime uniquement les doublons *adjacents*. Pour éliminer tous les doublons, vous avez trois options : trier le vecteur avant d'appeler, déplacer les données dans un ensemble, ou (pour conserver les éléments dans leur ordre d'origine) utiliser cette astuce : `'s' .dedup() .retain()`

```
let mut byte_vec = b"Mississsssissippi".to_vec();

let mut seen = HashSet::new();
byte_vec.retain(|r| seen.insert(*r));

assert_eq!(&byte_vec, b"Misp");
```

Cela fonctionne car il retourne lorsque l'ensemble contient déjà l'élément que nous insérons. `.insert() false`

`vec.dedup_by(same)`

La même chose que , mais il utilise la fonction ou la fermeture , au lieu de l'opérateur, pour vérifier si deux éléments doivent être considérés comme égaux. `vec.dedup() same(&mut elem1, &mut elem2) ==`

`vec.dedup_by_key(key)`

La même chose que , mais il traite deux éléments comme égaux si `.vec.dedup() key(&mut elem1) == key(&mut elem2)`

Par exemple, si est un , vous pouvez écrire : `errors Vec<Box<dyn Error>>`

```
// Remove errors with redundant messages.
errors.dedup_by_key(|err| err.to_string());
```

De toutes les méthodes abordées dans cette section, seules les valeurs clonent. Les autres travaillent en déplaçant les valeurs d'un endroit à un autre. `.resize()`

Joignant

Deux méthodes fonctionnent sur *des tableaux de tableaux, par lesquels* nous entendons tout tableau, tranche ou vecteur dont les éléments sont eux-mêmes des tableaux, des tranches ou des vecteurs :

`slices.concat()`

Renvoie un nouveau vecteur créé en concaténant toutes les tranches :

```
assert_eq!([[1, 2], [3, 4], [5, 6]].concat(),
           vec![1, 2, 3, 4, 5, 6]);
```

`slices.join(&separator)`

La même chose, sauf qu'une copie de la valeur est insérée entre les tranches : `separator`

```
assert_eq!([[1, 2], [3, 4], [5, 6]].join(&0),
           vec![1, 2, 0, 3, 4, 0, 5, 6]);
```

Fractionnement

Il est facile d'obtenir de nombreuses non-références dans un tableau, une tranche ou un vecteur à la fois : `mut`

```
let v = vec![0, 1, 2, 3];
let a = &v[i];
let b = &v[j];

let mid = v.len() / 2;
```

```
let front_half = &v[..mid];
let back_half = &v[mid..];
```

Obtenir plusieurs références n'est pas si facile: `mut`

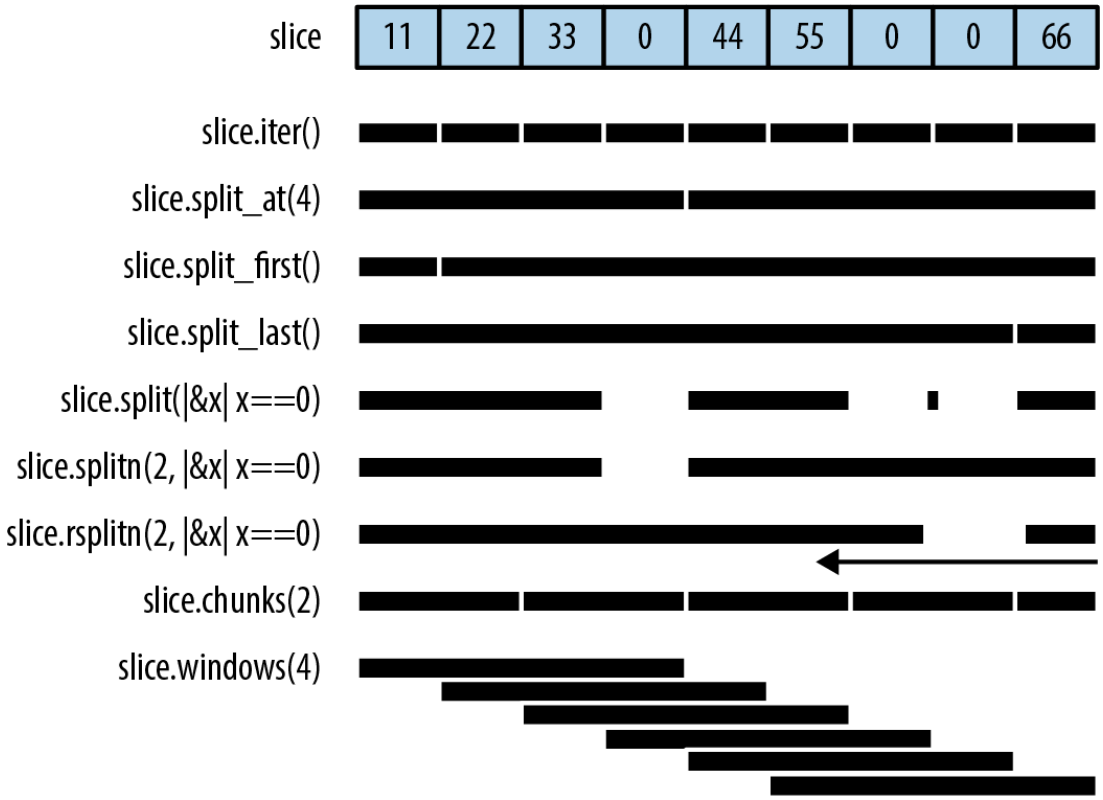
```
let mut v = vec![0, 1, 2, 3];
let a = &mut v[i];
let b = &mut v[j];  // error: cannot borrow `v` as mutable
                    //        more than once at a time

*a = 6;             // references `a` and `b` get used here,
*b = 7;             // so their lifetimes must overlap
```

Rust l'interdit parce que si , alors et serait deux références au même entier, en violation des règles de sécurité de Rust. (Voir « Partage versus mutation ».) `i == j a b mut`

Rust a plusieurs méthodes qui peuvent emprunter des références à deux ou plusieurs parties d'un tableau, d'une tranche ou d'un vecteur à la fois. Contrairement au code précédent, ces méthodes sont sûres, car de par leur conception, elles divisent toujours les données en régions *sans chevauchement*. Beaucoup de ces méthodes sont également pratiques pour travailler avec des non-tranches, il existe donc des versions non-. `mut mut mut mut`

La figure 16-2 illustre ces méthodes.



Graphique 16-2. Méthodes de fractionnement illustrées (note: le petit rectangle dans la sortie de est une tranche vide causée par les deux séparateurs adjacents, et produit sa sortie dans l'ordre de bout en début, contrairement aux autres) `slice.split()` `rsplitn`

Aucune de ces méthodes ne modifie directement un tableau, une tranche ou un vecteur ; ils renvoient simplement de nouvelles références à des parties des données à l'intérieur:

`slice.iter()`, `slice.iter_mut()`

Produire une référence à chaque élément de . Nous les avons couverts dans « Itération ». `slice`

`slice.split_at(index)`, `slice.split_at_mut(index)`

Casser une tranche en deux, en renvoyant une paire. est équivalent à . Ces méthodes paniquent si est hors limites. `slice.split_at(index)` `(&slice[..index], &slice[index..])` `index`

`slice.split_first()`, `slice.split_first_mut()`

Renvoiez également une paire : une référence au premier élément () et une référence de tranche à tout le reste (). `slice[0]` `slice[1..]`

Le type de retour de est ; le résultat est si est vide. `.split_first()` `Option<(&T, &[T])>` `None` `slice`

`slice.split_last()`, `slice.split_last_mut()`

Ceux-ci sont analogues mais séparent le dernier élément plutôt que le premier.

Le type de retour de est . `.split_last()` `Option<(&T, &[T])>`

`slice.split(is_sep)`, `slice.split_mut(is_sep)`

Divisez en une ou plusieurs sous-couches, en utilisant la fonction ou la fermeture pour déterminer où diviser. Ils renvoient un itérateur sur les sous-couches. `slice is_sep`

Lorsque vous consommez l'itérateur, il appelle chaque élément de la tranche. Si est , l'élément est un séparateur. Les séparateurs ne sont inclus dans aucune sous-sous-section de sortie. `is_sep(&element) is_sep(&element) true`

La sortie contient toujours au moins une sous-section, plus une par séparateur. Les sous-couches vides sont incluses chaque fois que des séparateurs apparaissent adjacents les uns aux autres ou aux extrémités de . `slice`

`slice.split_inclusive(is_sep)`, `slice.split_inclusive_mut(is_sep)`

Ceux-ci fonctionnent comme et , mais incluent le séparateur à la fin de la sous-section précédente plutôt que de l'exclure. `split split_mut`

`slice.rsplit(is_sep)`, `slice.rsplit_mut(is_sep)`

Tout comme et , mais commencez à la fin de la tranche. `slice slice_mut`

`slice.splitn(n, is_sep)`, `slice.splitn_mut(n, is_sep)`

Les mêmes mais ils produisent au plus des sous-couches. Une fois les premières tranches trouvées, n'est pas appelé à nouveau. La dernière sous-sous-section contient tous les éléments restants. `n n-1 is_sep`

`slice.rsplitn(n, is_sep)`, `slice.rsplitn_mut(n, is_sep)`

Tout comme et sauf que la tranche est scannée dans l'ordre inverse. C'est-à-dire que ces méthodes se divisent sur les *derniers* séparateurs de la tranche, plutôt que sur le premier, et les sous-couches sont produites à partir de la fin. `.splitn() .splitn_mut() n-1`

`slice.chunks(n)`, `slice.chunks_mut(n)`

Renvoyer un itérateur sur des sous-sous-couches de longueur sans chevauchement . S'il ne se divise pas exactement, le dernier morceau contiendra moins d'éléments. `n n slice.len() n`

`slice.rchunks(n)`, `slice.rchunks_mut(n)`

Tout comme et , mais commencez à la fin de la tranche. `slice.chunks slice.chunks_mut`

`slice.chunks_exact(n)`, `slice.chunks_exact_mut(n)`

Renvoyer un itérateur sur des sous-sous-couches de longueur sans chevauchement . Si ne divise pas , le dernier morceau (avec moins d'éléments)

est disponible dans la méthode du résultat. `n n slice.len() n remainder()`

`slice.rchunks_exact(n), slice.rchunks_exact_mut(n)`

Tout comme et , mais commencez à la fin de la tranche. `slice.chunks_exact slice.chunks_exact_mut`

Il existe une autre méthode pour itérer sur les sous-couches :

`slice.windows(n)`

Renvoie un itérateur qui se comporte comme une « fenêtre coulissante » sur les données dans . Il produit des sous-couches qui couvrent des éléments consécutifs de . La première valeur produite est , la seconde est , et ainsi de suite. `slice n slice &slice[0..n] &slice[1..n+1]`

Si est supérieure à la longueur de , aucune tranche n'est produite. Si est 0, la méthode panique. `n slice n`

Par exemple, si , alors nous pouvons produire toutes les périodes de sept jours en appelant . `days.len() == 31 days days.windows(7)`

Une fenêtre coulissante de taille 2 est pratique pour explorer comment une série de données change d'un point de données à l'autre :

```
let changes = daily_high_temperatures
                  .windows(2)              // get adjacent days' tem
                  .map(|w| w[1] - w[0])    // how much did it change
                  .collect::<Vec<_>>();
```

Étant donné que les sous-sous-couches se chevauchent, il n'existe aucune variante de cette méthode qui renvoie des références. `mut`

Échange

Il existe des méthodes pratiques pour échanger le contenu des tranches :

`slice.swap(i, j)`

Échange les deux éléments et . `slice[i] slice[j]`

`slice_a.swap(&mut slice_b)`

Échange l'intégralité du contenu de et . et doit avoir la même longueur. `slice_a slice_b slice_a slice_b`

Les vecteurs ont une méthode connexe pour supprimer efficacement n'importe quel élément:

`vec.swap_remove(i)`

Supprime et renvoie . C'est comme si, sauf qu'au lieu de faire glisser le reste des éléments du vecteur pour combler l'écart, il déplace simplement le dernier élément de l'espace dans l'espace. C'est utile lorsque vous ne vous souciez pas de l'ordre des éléments laissés dans le vecteur. `vec[i] vec.remove(i) vec`

Remplissage

Il existe deux méthodes pratiques pour remplacer le contenu des tranches modifiables :

`slice.fill(value)`

Remplit la tranche avec des clones de . `value`

`slice.fill_with(function)`

Remplit la tranche avec des valeurs créées en appelant la fonction donnée. Ceci est particulièrement utile pour les types qui implémentent , mais ne sont pas , comme ou quand n'est pas . `Default Clone Option<T> Vec<T> T Clone`

Tri et recherche

Les tranches offrent trois méthodes de tri :

`slice.sort()`

Trie les éléments dans un ordre croissant. Cette méthode est présente uniquement lorsque le type d'élément implémente . `Ord`

`slice.sort_by(cmp)`

Trie les éléments de l'utilisation d'une fonction ou d'une fermeture pour spécifier l'ordre de tri. doit mettre en œuvre . `slice cmp cmp Fn(&T, &T) -> std::cmp::Ordering`

La mise en œuvre manuelle est pénible, sauf si vous déléguez à une méthode : `cmp .cmp()`

```
students.sort_by(|a, b| a.last_name.cmp(&b.last_name));
```

Pour trier par un champ, en utilisant un deuxième champ comme bris d'égalité, comparez les tuples :

```
students.sort_by(|a, b| {
    let a_key = (&a.last_name, &a.first_name);
    let b_key = (&b.last_name, &b.first_name);
    a_key.cmp(&b_key)
});
```

`slice.sort_by_key(key)`

Trie les éléments de dans l'ordre croissant par une clé de tri, donnée par la fonction ou la fermeture . Le type de doit implémenter où `.slice key key Fn(&T) -> K K: Ord`

Ceci est utile lorsqu'il contient un ou plusieurs champs ordonnés, de sorte qu'il peut être trié de plusieurs façons: `T`

```
// Sort by grade point average, lowest first.
students.sort_by_key(|s| s.grade_point_average());
```

Notez que ces valeurs de clé de tri ne sont pas mises en cache pendant le tri, de sorte que la fonction peut être appelée plus de *n* fois. `key`

Pour des raisons techniques, ne peut pas renvoyer de références empruntées à l'élément. Cela ne fonctionnera pas : `key(element)`

```
students.sort_by_key(|s| &s.last_name);  // error: can't infer life
```

Rust ne peut pas comprendre les durées de vie. Mais dans ces cas, il est assez facile de se rabattre sur . `.sort_by()`

Les trois méthodes effectuent un tri stable.

Pour trier dans l'ordre inverse, vous pouvez utiliser avec une fermeture qui permute les deux arguments. Prendre des arguments plutôt que de produire efficacement l'ordre inverse. Ou, vous pouvez simplement appeler la méthode après le tri: `sort_by cmp |b, a| |a, b| .reverse()`

`slice.reverse()`

Inverse une tranche en place.

Une fois qu'une tranche est triée, elle peut être recherchée efficacement :

`slice.binary_search(&value),,`
`slice.binary_search_by(&value,`
`cmp) slice.binary_search_by_key(&value, key)`

Toutes les recherches dans le fichier trié donné . Notez que c'est passé par référence. `value slice value`

Le type de retour de ces méthodes est . Ils renvoient si égaux dans l'ordre de tri spécifié. S'il n'y a pas un tel index, ils renvoient de telle sorte que l'insertion de à préserverait l'ordre. `Result<usize, usize> Ok(index) slice[index] value Err(insertion_point) value insertion_point`

Bien sûr, une recherche binaire ne fonctionne que si la tranche est en fait triée dans l'ordre spécifié. Sinon, les résultats sont arbitraires : garbage in, garbage out.

Depuis et ont des valeurs NaN, ils ne s'implémentent pas et ne peuvent pas être utilisés directement comme clés avec les méthodes de tri et de recherche binaire. Pour obtenir des méthodes similaires qui fonctionnent sur des données à virgule flottante, utilisez la caisse. `f32 f64 Ord ord_subset`

Il existe une méthode pour rechercher un vecteur qui n'est pas trié :

`slice.contains(&value)`

Renvoie si un élément de est égal à . Cela vérifie simplement chaque élément de la tranche jusqu'à ce qu'une correspondance soit trouvée. Encore une fois, est passé par référence. `true slice value value`

Pour trouver l'emplacement d'une valeur dans une tranche, comme en JavaScript, utilisez un itérateur : `array.indexOf(value)`

```
slice.iter().position(|x| *x == value)
```

Cela renvoie un fichier . `Option<usize>`

Comparaison des tranches

Si un type prend en charge les opérateurs et (le trait décrit dans « Comparaisons d'équivalence »), les tableaux, les tranches et les vecteurs les prennent également en charge. Deux tranches sont égales si elles ont la même longueur et que leurs éléments correspondants sont égaux. Il en va de même pour les tableaux et les vecteurs. `T == != PartialEq [T; N] [T] Vec<T>`

Si prend en charge les opérateurs , , , et (le trait, décrit dans « Comparaisons ordonnées »), alors les tableaux, les tranches et les vecteurs de faire

aussi. Les comparaisons de tranches sont lexicographiques. `T < <= > >= PartialOrd T`

Deux méthodes pratiques effectuent des comparaisons de tranches courantes :

`slice.starts_with(other)`

Renvoie si commence par une séquence de valeurs égales aux éléments de la tranche : `true slice other`

```
assert_eq!([1, 2, 3, 4].starts_with(&[1, 2]), true);
assert_eq!([1, 2, 3, 4].starts_with(&[2, 3]), false);
```

`slice.ends_with(other)`

Similaire mais vérifie la fin de : `slice`

```
assert_eq!([1, 2, 3, 4].ends_with(&[3, 4]), true);
```

Éléments aléatoires

Les nombres aléatoires ne sont pas intégrés à la bibliothèque standard Rust. La caisse, qui les fournit, offre ces deux méthodes pour obtenir une sortie aléatoire à partir d'un tableau, d'une tranche ou d'un vecteur : `rand`

`slice.choose(&mut rng)`

Renvoie une référence à un élément aléatoire d'une tranche. Comme et , cela renvoie un qui n'est que si la tranche est vide. `slice.first() slice.last() Option<&T> None`

`slice.shuffle(&mut rng)`

Réorganise de manière aléatoire les éléments d'une tranche en place. La tranche doit être passée par référence. `mut`

Ce sont des méthodes du trait, vous avez donc besoin d'un , un générateur de nombres aléatoires, afin de les appeler. Heureusement, il est facile d'en obtenir un en appelant . Pour mélanger le vecteur , nous pouvons écrire: `rand::Rng Rng rand::thread_rng() my_vec`

```
use rand::seq::SliceRandom;
use rand::thread_rng;

my_vec.shuffle(&mut thread_rng());
```

Rust exclut les erreurs d'invalidation

La plupart des langages de programmation traditionnels ont des collections et des itérateurs, et ils ont tous une variante de cette règle : ne modifiez pas une collection pendant que vous itérez dessus. Par exemple, l'équivalent Python d'un vecteur est une liste :

```
my_list = [1, 3, 5, 7, 9]
```

Supposons que nous essayions de supprimer toutes les valeurs supérieures à 4 de : `my_list`

```
for index, val in enumerate(my_list):
    if val > 4:
        del my_list[index]  # bug: modifying list while iterating

print(my_list)
```

(La fonction est l'équivalent de Python de la méthode de Rust, décrite dans « énumérer ».) `enumerate .enumerate()`

Ce programme, étonnamment, imprime . Mais sept est plus grand que quatre. Comment cela s'est-il glissé? Il s'agit d'une erreur d'invalidation : le programme modifie les données tout en itérant dessus, *invalidant* l'itérateur. En Java, le résultat serait une exception ; en C++, il s'agit d'un comportement non défini. En Python, bien que le comportement soit bien défini, il n'est pas intuitif : l'itérateur saute un élément. n'est jamais . `[1, 3, 7] val 7`

Essayons de reproduire ce bug dans Rust:

```
fn main() {
    let mut my_vec = vec![1, 3, 5, 7, 9];

    for (index, &val) in my_vec.iter().enumerate() {
        if val > 4 {
            my_vec.remove(index);  // error: can't borrow `my_vec` as m
        }
    }
    println!("{:?}", my_vec);
}
```

Naturellement, Rust rejette ce programme au moment de la compilation. Lorsque nous appelons , il emprunte une référence partagée (non-) au

vecteur. La référence vit aussi longtemps que l'itérateur, jusqu'à la fin de la boucle. Nous ne pouvons pas modifier le vecteur en appelant tant qu'une non-référence
existe. `my_vec.iter()` `mut for` `my_vec.remove(index)` `mut`

Avoir une erreur qui vous est signalée est bien, mais bien sûr, vous devez toujours trouver un moyen d'obtenir le comportement souhaité! La solution la plus simple ici est d'écrire:

```
my_vec.retain(|&val| val <= 4);
```

Ou, vous pouvez faire ce que vous feriez en Python ou dans n'importe quel autre langage : créer un nouveau vecteur à l'aide d'un fichier
`.filter`

VecDeque<T>

`Vec` prend en charge l'ajout et la suppression efficaces d'éléments uniquement à la fin. Lorsqu'un programme a besoin d'un endroit pour stocker des valeurs qui « font la queue », cela peut être lent. `Vec`

Rust's est un *deque* (prononcé « deck »), une file d'attente à double extrémité. Il prend en charge les opérations d'ajout et de suppression efficaces à l'avant et à l'arrière: `std::collections::VecDeque<T>`

`deque.push_front(value)`

Ajoute une valeur à l'avant de la file d'attente.

`deque.push_back(value)`

Ajoute une valeur à la fin. (Cette méthode est utilisée beaucoup plus que , car la convention habituelle pour les files d'attente est que les valeurs sont ajoutées à l'arrière et supprimées à l'avant, comme les personnes qui attendent dans une file.) `.push_front()`

`deque.pop_front()`

Supprime et renvoie la valeur frontale de la file d'attente, en renvoyant un fichier si la file d'attente est vide, par exemple . `Option<T>` `None` `vec.pop()`

`deque.pop_back()`

Supprime et renvoie la valeur à l'arrière, en renvoyant à nouveau un fichier
. `Option<T>`

`deque.front(), deque.back()`

Travailler comme et . Ils renvoient une référence à l'élément avant ou arrière de la file d'attente. La valeur renvoyée est un si la file d'attente est vide. `vec.first()` `vec.last()` `Option<&T>` `None`

`deque.front_mut()`, `deque.back_mut()`

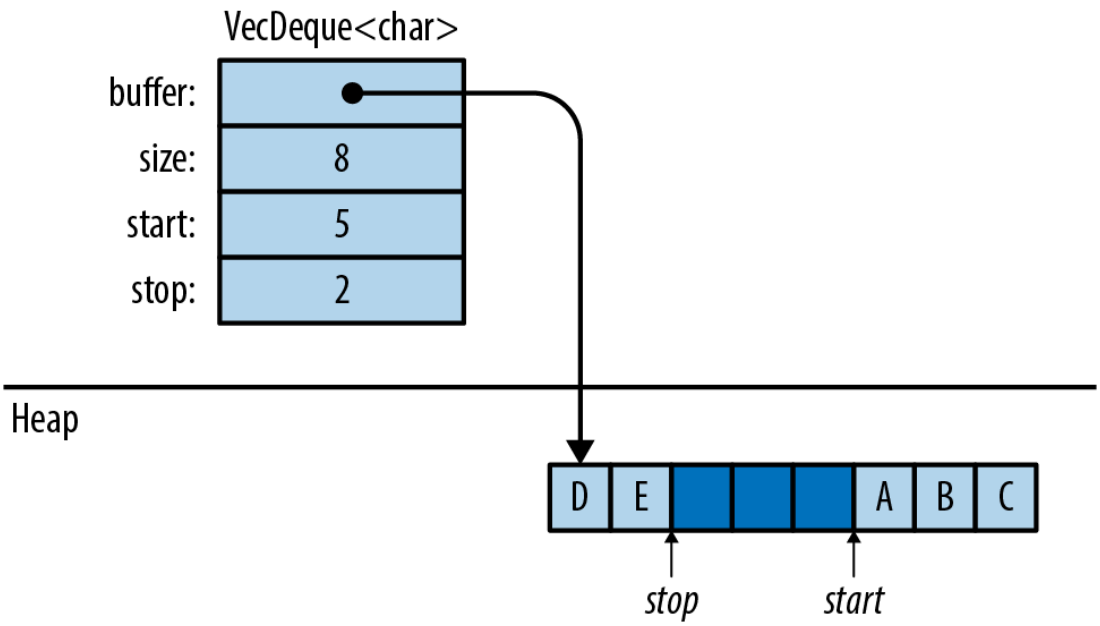
Travailler comme et , revenir . `vec.first_mut()` `vec.last_mut()` `Option<&mut T>`

L'implémentation de est un tampon en anneau, comme le montre la figure 16-3. `VecDeque`

Comme un , il a une allocation de tas unique où les éléments sont stockés. Contrairement à , les données ne commencent pas toujours au début de cette région, et elles peuvent « envelopper » la fin, comme indiqué. Les éléments de ce deque, dans l'ordre, sont . a des champs privés, étiquetés et dans la figure, qu'il utilise pour se souvenir de l'endroit où dans la mémoire tampon les données commencent et se terminent. `Vec` `Vec ['A', 'B', 'C', 'D', 'E']` `VecDeque` `start` `stop`

Ajouter une valeur à la file d'attente, à chaque extrémité, signifie réclamer l'un des emplacements inutilisés, illustré comme les blocs les plus sombres, envelopper ou allouer une plus grande partie de la mémoire si nécessaire.

`VecDeque` gère l'emballage, de sorte que vous n'avez pas à y penser. La figure 16-3 est une vue des coulisses de la façon dont Rust fait vite. `.pop_front()`



Graphique 16-3. Comment a est stocké en mémoire `VecDeque`

Souvent, lorsque vous avez besoin d'un deque, et sont les deux seules méthodes dont vous aurez besoin. Les fonctions associées au type et

`VecDeque::with_capacity(n)`, pour créer des files d'attente, sont comme leurs homologues dans `Vec`. De nombreuses méthodes sont également implémentées pour `VecDeque` : et, `.insert(index, value)`, `.remove(index)`, `.extend(iterable)`, etc. `.push_back()` `.pop_front()` `VecDeque::new()` `Vec` `.len()` `.is_empty()`

Les deques, comme les vecteurs, peuvent être itérés par valeur, par référence partagée ou par référence. Ils ont les trois méthodes d'itération , et . Ils peuvent être indexés de la manière habituelle : `.mut` `.into_iter()` `.iter()` `.iter_mut()` `deque[index]`

Parce que les deques ne stockent pas leurs éléments de manière contiguë dans la mémoire, ils ne peuvent pas hériter de toutes les méthodes de tranches. Mais si vous êtes prêt à payer le coût du déplacement du contenu, fournit une méthode qui résoudra ce problème: `VecDeque`

`deque.make_contiguous()`

Prend et réorganise le dans la mémoire contiguë, en renvoyant . `&mut self` `VecDeque` `&mut [T]`

`Vec` s et s sont étroitement liés, et la bibliothèque standard fournit deux implémentations de traits pour une conversion facile entre les deux : `VecDeque`

`Vec::from(deque)`

`Vec<T>` implémente , donc cela transforme un deque en vecteur. Cela coûte du temps O(n), car cela peut nécessiter de réorganiser les éléments. `From<VecDeque<T>>`

`VecDeque::from(vec)`

`VecDeque<T>` implémente , donc cela transforme un vecteur en deque. C'est aussi O(n), mais c'est généralement rapide, même si le vecteur est grand, car l'allocation de tas du vecteur peut simplement être déplacée vers la nouvelle deque. `From<Vec<T>>`

Cette méthode facilite la création d'une deque avec des éléments spécifiés, même s'il n'existe pas de macro standard : `vec_deque![]`

```
use std::collections::VecDeque;

let v = VecDeque::from(vec![1, 2, 3, 4]);
```

BinaryHeap<T>

A est une collection dont les éléments sont maintenus vaguement organisés de sorte que la plus grande valeur bouillonne toujours jusqu'à l'avant de la file d'attente. Voici les trois méthodes les plus couramment utilisées : `BinaryHeap BinaryHeap`

`heap.push(value)`

Ajoute une valeur au tas.

`heap.pop()`

Supprime et renvoie la plus grande valeur du tas. Il renvoie un c'est-à-dire si le tas était vide. `Option<T> None`

`heap.peek()`

Renvoie une référence à la valeur la plus élevée du tas. Le type de retour est . `Option<&T>`

`heap.peek_mut()`

Renvoie un , qui agit comme une référence modifiable à la plus grande valeur du tas et fournit la fonction associée au type pour extraire cette valeur du tas. En utilisant cette méthode, nous pouvons choisir de pop ou non pop du tas en fonction de la valeur maximale: `PeekMut<T> pop()`

```
use std::collections::binary_heap::PeekMut;

if let Some(top) = heap.peek_mut() {
    if *top > 10 {
        PeekMut::pop(top);
    }
}
```

`BinaryHeap` prend également en charge un sous-ensemble des méthodes sur , y compris , , , , `.clear()` et . `Vec BinaryHeap::new() .len() .is_empty() .capacity() .append(&mut heap2)`

Par exemple, supposons que nous remplissions a avec un tas de nombres : `BinaryHeap`

```
use std::collections::BinaryHeap;

let mut heap = BinaryHeap::from(vec![2, 3, 8, 6, 9, 5, 4]);
```

La valeur se trouve en haut du tas : `9`

```
assert_eq!(heap.peek(), Some(&9));
assert_eq!(heap.pop(), Some(9));
```

La suppression de la valeur réorganise également légèrement les autres éléments de sorte qu'ils soient maintenant à l'avant, et ainsi de suite : `9 8`

```
assert_eq!(heap.pop(), Some(8));
assert_eq!(heap.pop(), Some(6));
assert_eq!(heap.pop(), Some(5));
...
```

Bien sûr, ne se limite pas aux chiffres. Il peut contenir n'importe quel type de valeur qui implémente le trait intégré. `BinaryHeap Ord`

Cela rend utile en tant que file d'attente de travail. Vous pouvez définir une structure de tâche qui s'implémente sur la base de la priorité afin que les tâches de priorité plus élevée soient plus importantes que les tâches de priorité inférieure. Ensuite, créez un pour contenir toutes les tâches en attente. Sa méthode renverra toujours l'élément le plus important, la tâche sur laquelle votre programme devrait travailler ensuite. `BinaryHeap Ord Greater BinaryHeap .pop()`

Remarque: est itérable, et il a une méthode, mais les itérateurs produisent les éléments du tas dans un ordre arbitraire, pas du plus grand au plus petit. Pour consommer les valeurs d'un ordre de priorité, utilisez une boucle : `BinaryHeap .iter() BinaryHeap while`

```
while let Some(task) = heap.pop() {
    handle(task);
}
```

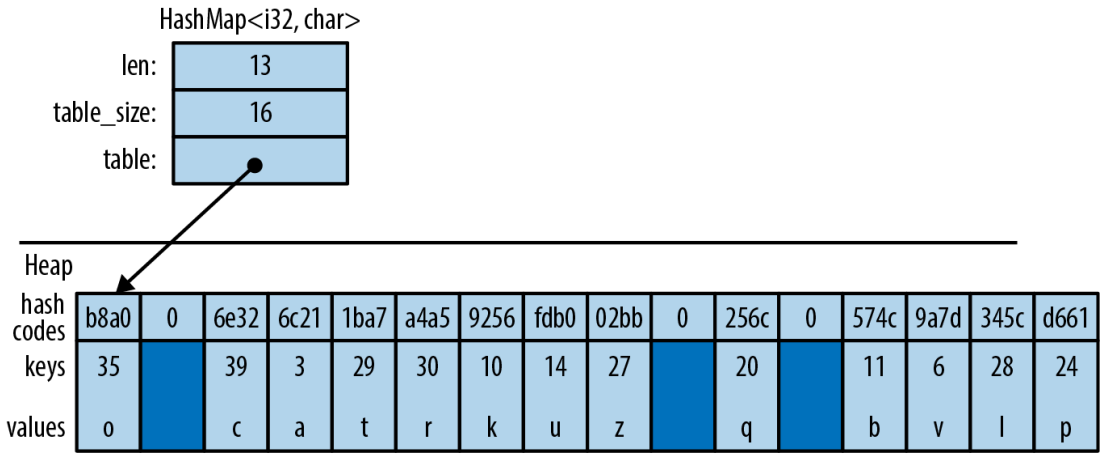
HashMap<K, V> et BTreeMap<K, V>

Une *carte* est une collection de paires clé-valeur (*appelées entrées*). Il n'y a pas deux entrées qui ont la même clé, et les entrées sont maintenues organisées de sorte que si vous avez une clé, vous pouvez rechercher efficacement la valeur correspondante dans une carte. En bref, une carte est une table de choix.

Rust propose deux types de cartes : et . Les deux partagent bon nombre des mêmes méthodes; la différence réside dans la façon dont les deux entrées de conservation sont organisées pour une recherche rapide. `HashMap<K, V>` `BTreeMap<K, V>`

A stocke les clés et les valeurs dans une table de hachage, il nécessite donc un type de clé qui implémente et , les traits standard pour le hachage et l'égalité. `HashMap` `K` `Hash` `Eq`

La figure 16-4 montre comment a est organisé en mémoire. Les régions plus sombres ne sont pas utilisées. Toutes les clés, valeurs et codes de hachage mis en cache sont stockés dans une seule table allouée au tas. L'ajout d'entrées finit par forcer l'allocation d'une table plus grande et à y déplacer toutes les données. `HashMap` `HashMap`

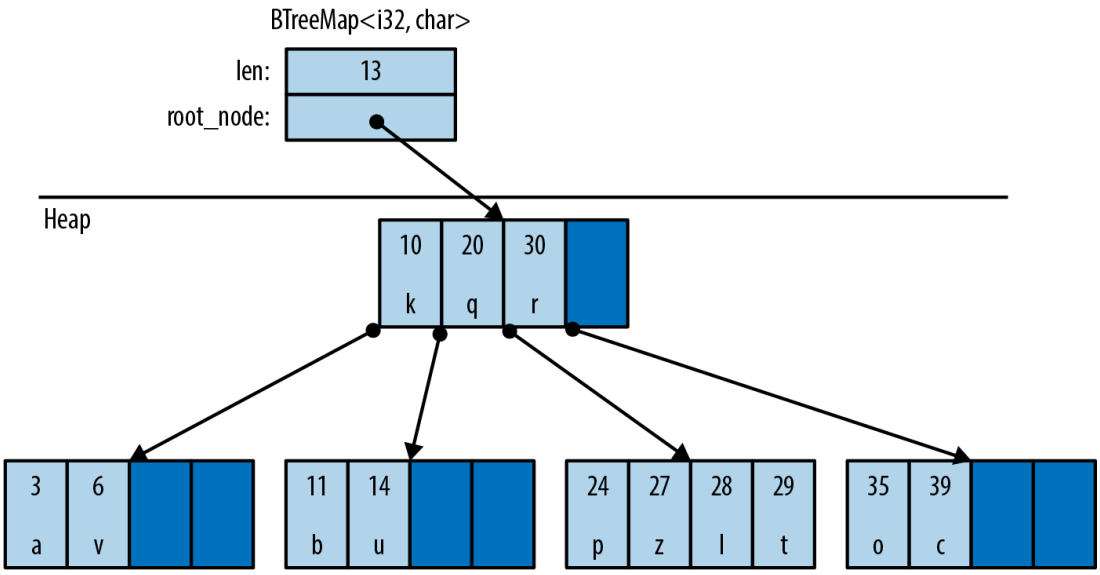


Graphique 16-4. A en mémoire `HashMap`

A stocke les entrées dans l'ordre par clé, dans une arborescence, de sorte qu'il nécessite un type de clé qui implémente . La figure 16-5 montre un fichier . Encore une fois, les régions les plus sombres sont des capacités inutilisées. `BTreeMap` `K` `Ord` `BTreeMap`

A stocke ses entrées dans des *nœuds*. La plupart des nœuds d'un ne contiennent que des paires clé-valeur. Les nœuds non feuilles, comme le nœud racine illustré dans cette figure, ont également de la place pour les pointeurs vers les nœuds enfants. Pointeur entre et pointe vers un nœud enfant contenant des clés entre et . L'ajout d'entrées nécessite souvent de faire glisser certaines des entrées existantes d'un nœud vers la droite, pour les garder triées, et implique parfois l'allocation de nouveaux nœuds. `BTreeMap` `BTreeMap` `(20, 'q')` `(30, 'r')` `20` `30`

Cette image est un peu simplifiée pour tenir sur la page. Par exemple, les nœuds réels ont de la place pour 11 entrées, pas 4. `BTreeMap`



Graphique 16-5. A en mémoire `BTreeMap`

La bibliothèque standard Rust utilise des arbres B plutôt que des arbres binaires équilibrés, car les arbres B sont plus rapides sur le matériel moderne. Un arbre binaire peut utiliser moins de comparaisons par recherche qu'un arbre B, mais la recherche d'un arbre B a une meilleure *localité*, c'est-à-dire que les accès à la mémoire sont regroupés plutôt que dispersés sur l'ensemble du tas. Cela rend les erreurs de cache cpu plus rares. C'est une augmentation significative de la vitesse.

Il existe plusieurs façons de créer une carte :

`HashMap::new()`, `BTreeMap::new()`

Créez de nouvelles cartes vides.

`iter.collect()`

Peut être utilisé pour créer et remplir une nouvelle paire clé-valeur ou à partir de paires clé-valeur. doit être un fichier
. `HashMap BTreeMap iter Iterator<Item=(K, V)>`

`HashMap::with_capacity(n)`

Crée une nouvelle carte de hachage vide avec de la place pour au moins *n* entrées. s, comme les vecteurs, stockent leurs données dans une seule allocation de tas, de sorte qu'ils ont une capacité et les méthodes associées , et . ne le font pas. `HashMap hash_map.capacity() hash_map.reserve(additional) hash_map.shrink_to_fit() BTreeMap`

`HashMap` s et s ont les mêmes méthodes de base pour travailler avec des clés et des valeurs : `BTreeMap`

`map.len()`

Renvoie le nombre d'entrées.

`map.is_empty()`

Renvoie s'il n'y a pas d'entrées. `true map`

`map.contains_key(&key)`

Renvoie si la carte comporte une entrée pour le fichier . `true key`

`map.get(&key)`

Recherche une entrée avec le fichier . Si une entrée correspondante est trouvée, celle-ci renvoie , où est une référence à la valeur correspondante. Sinon, cela renvoie . `map key Some(r) r None`

`map.get_mut(&key)`

Similaire, mais il renvoie une référence à la valeur. `mut`

En général, les cartes vous permettent d'avoir accès aux valeurs qui y sont stockées, mais pas aux clés. Les valeurs sont à vous de modifier comme vous le souhaitez. Les clés appartiennent à la carte elle-même; il doit s'assurer qu'ils ne changent pas, car les entrées sont organisées par leurs clés. Modifier une clé en place serait un bug. `mut`

`map.insert(key, value)`

Insère l'entrée et renvoie l'ancienne valeur, le cas échéant. Le type de retour est . S'il y a déjà une entrée pour dans la carte, la nouvelle entrée écrase l'ancienne. `(key, value) map Option<V> key value`

`map.extend(iterable)`

Itére sur les éléments de et insère chacune de ces paires clé-valeur dans . `(K, V) iterable map`

`map.append(&mut map2)`

Déplace toutes les entrées de dans . Après, est vide. `map2 map map2`

`map.remove(&key)`

Recherche et supprime toute entrée avec la valeur donnée de , renvoyant la valeur supprimée, le cas échéant. Le type de retour est . `key map Option<V>`

`map.remove_entry(&key)`

Recherche et supprime toute entrée avec la valeur donnée de , renvoyant la clé et la valeur supprimées, le cas échéant. Le type de retour est . `key map Option<(K, V)>`

`map.retain(test)`

Supprime tous les éléments qui ne réussissent pas le test donné. L'argument est une fonction ou une fermeture qui implémente . Pour chaque élément de , cet appel , et s'il retourne , l'élément est supprimé de la carte et supprimé. `test FnMut(&K, &mut V) -> bool map test(&key, &mut value) false`

En dehors de la performance, c'est comme écrire :

```
map = map.into_iter().filter(test).collect();
```

`map.clear()`

Supprime toutes les entrées.

Une carte peut également être interrogée entre crochets : . C'est-à-dire que les cartes implémentent le trait intégré. Cependant, cela panique s'il n'y a pas déjà une entrée pour le donné , comme un accès au tableau hors limites, alors utilisez cette syntaxe uniquement si l'entrée que vous recherchez est sûre d'être remplie. `map[&key] Index key`

L'argument de , , , et n'a pas besoin d'avoir le type exact . Ces méthodes sont génériques sur les types qui peuvent être empruntés à . Il est correct d'appeler un , même si ce n'est pas exactement un , car implémente . Pour plus de détails, voir « Emprunter et empruntermut ». `key .contains_key() .get() .get_mut() .remove() &K K fish_map.contains_key("conger") HashMap<String, Fish> "conger" String String Borrow<&str>`

Étant donné que a conserve ses entrées triées par clé, il prend en charge une opération supplémentaire : `BTreeMap<K, V>`

`btree_map.split_off(&key)`

Se divise en deux. Entrées dont les touches sont inférieures à celles qui restent dans . Renvoie un nouveau contenant les autres entrées. `btree_map key btree_map BTreeMap<K, V>`

Entrées

Les deux et ont un type correspondant. Le but des entrées est d'éliminer les recherches de carte redondantes. Par exemple, voici du code pour obtenir ou créer un enregistrement étudiant : `HashMap BTreeMap Entry`

```
// Do we already have a record for this student?
if !student_map.contains_key(name) {
    // No: create one.
    student_map.insert(name.to_string(), Student::new());
}
// Now a record definitely exists.
let record = student_map.get_mut(name).unwrap();
...
```

Cela fonctionne bien, mais il accède deux ou trois fois, en faisant la même recherche à chaque fois. `student_map`

L'idée avec les entrées est que nous ne faisons la recherche qu'une seule fois, produisant une valeur qui est ensuite utilisée pour toutes les opérations ultérieures. Ce one-liner est équivalent à tout le code précédent, sauf qu'il ne fait la recherche qu'une seule fois : `Entry`

```
let record = student_map.entry(name.to_string()).or_insert_with(Student
```

La valeur renvoyée par agit comme une référence modifiable à un endroit de la carte qui est soit *occupé* par une paire clé-valeur, soit *vacant*, ce qui signifie qu'il n'y a pas encore d'entrée. Si elle est vacante, la méthode de l'entrée insère un nouveau fichier . La plupart des utilisations des entrées sont comme ceci: court et
doux. `Entry student_map.entry(name.to_string()) .or_insert_w ith() Student`

Toutes les valeurs sont créées par la même méthode : `Entry`

`map.entry(key)`

Renvoie un pour le fichier . S'il n'y a pas de telle clé dans la carte, cela renvoie un fichier vacant . `Entry key Entry`

Cette méthode prend son argument par référence et renvoie un avec une durée de vie correspondante : `self mut Entry`

```
pub fn entry<'a>(&'a mut self, key: K) -> Entry<'a, K, V>
```

Le type a un paramètre de durée de vie car il s'agit en fait d'une sorte de référence empruntée à la carte. Tant qu'il existe, il a un accès exclusif à la carte. `Entry 'a mut Entry`

De retour dans « Structs Containing References », nous avons vu comment stocker des références dans un type et comment cela affecte les durées de vie. Maintenant, nous voyons à quoi cela ressemble du point de vue de l'utilisateur. C'est ce qui se passe avec . `Entry`

Malheureusement, il n'est pas possible de passer une référence de type à cette méthode si la carte comporte des clés. La méthode, dans ce cas, nécessite un réel . `&str String .entry() String`

`Entry` les valeurs fournissent trois méthodes pour traiter les entrées vacantes :

`map.entry(key).or_insert(value)`

S'assure qu'il contient une entrée avec le donné, en insérant une nouvelle entrée avec le donné si nécessaire. Il renvoie une référence à la valeur nouvelle ou existante. `map key value mut`

Supposons que nous devions compter les votes. Nous pouvons écrire :

```
let mut vote_counts: HashMap<String, usize> = HashMap::new();
for name in ballots {
    let count = vote_counts.entry(name).or_insert(0);
    *count += 1;
}
```

`.or_insert()` renvoie une référence, de sorte que le type de est `.mut count &mut usize`

`map.entry(key).or_default()`

Garantit qu'il contient une entrée avec la clé donnée, en insérant une nouvelle entrée avec la valeur renvoyée par si nécessaire. Cela ne fonctionne que pour les types qui implémentent . Par exemple, cette méthode renvoie une référence à la valeur nouvelle ou existante. `map Default::default() Default or_insert mut`

`map.entry(key).or_insert_with(default_fn)`

C'est la même chose, sauf que s'il doit créer une nouvelle entrée, il appelle pour produire la valeur par défaut. S'il y a déjà une entrée pour dans le , alors n'est pas utilisé. `default_fn() key map default_fn`

Supposons que nous voulions savoir quels mots apparaissent dans quels fichiers. Nous pouvons écrire :

```
// This map contains, for each word, the set of files it appears in
let mut word_occurrence: HashMap<String, HashSet<String>> =
    HashMap::new();
for file in files {
    for word in read_words(file)? {
        let set = word_occurrence
            .entry(word)
            .or_insert_with(HashSet::new);
        set.insert(file.clone());
```

```
        }
    }
```

`Entry` fournit également un moyen pratique de modifier uniquement les champs existants.

`map.entry(key).and_modify(closure)`

Appelle si une entrée avec la clé existe, en passant une référence modifiable à la valeur. Il renvoie le , afin qu'il puisse être enchaîné avec d'autres méthodes. `closure key Entry`

Par exemple, nous pourrions l'utiliser pour compter le nombre d'occurrences de mots dans une chaîne :

```
// This map contains all the words in a given string,
// along with the number of times they occur.
let mut word_frequency: HashMap<&str, u32> = HashMap::new();
for c in text.split_whitespace() {
    word_frequency.entry(c)
        .and_modify(|count| *count += 1)
        .or_insert(1);
}
```

Le type est un enum, défini ainsi pour (et de même pour) : `Entry HashMap BTreeMap`

```
// (in std::collections::hash_map)
pub enum Entry<'a, K, V> {
    Occupied(OccupiedEntry<'a, K, V>),
    Vacant(VacantEntry<'a, K, V>)
}
```

Les et types ont des méthodes pour insérer, supprimer et accéder aux entrées sans répéter la recherche initiale. Vous pouvez les trouver dans la documentation en ligne. Les méthodes supplémentaires peuvent parfois être utilisées pour éliminer une recherche redondante ou deux, mais et couvrir les cas courants. `OccupiedEntry VacantEntry .or_insert() .or_insert_with()`

Itération de carte

Il existe plusieurs façons d'itérer sur une carte :

- L'itération par valeur () produit des paires. Cela consomme la carte. `for (k, v) in map (K, V)`
- L'itération sur une référence partagée () produit des paires. `for (k, v) in &map (&K, &V)`
- L'itération sur une référence () produit des paires. (Encore une fois, il n'y a aucun moyen d'accéder aux clés stockées dans une carte, car les entrées sont organisées par leurs clés.) `mut for (k, v) in &mut map (&K, &mut V) mut`

Comme les vecteurs, les cartes ont des méthodes qui renvoient des itérateurs par référence, tout comme l'itération sur ou . De plus,: `.iter() .iter_mut() &map &mut map`

`map.keys()`

Renvoie un itérateur sur les seules clés, par référence.

`map.values()`

Renvoie un itérateur sur les valeurs, par référence.

`map.values_mut()`

Renvoie un itérateur sur les valeurs, par référence. `mut`

`map.into_iter(),, map.into_keys() map.into_values()`

Consommez la carte, en renvoyant un itérateur sur des tuples de clés et de valeurs, de clés ou de valeurs, respectivement. `(K, V)`

Tous les itérateurs visitent les entrées de la carte dans un ordre arbitraire. les itérateurs les visitent dans l'ordre par clé. `HashMap BTreeMap`

HashSet<T> et BTreeSet<T>

Les ensembles sont des collections de valeurs organisées pour un test d'appartenance rapide :

```
let b1 = large_vector.contains(&"needle");    // slow, checks every ele
let b2 = large_hash_set.contains(&"needle");  // fast, hash lookup
```

Un ensemble ne contient jamais plusieurs copies de la même valeur.

Les cartes et les ensembles ont des méthodes différentes, mais dans les coulisses, un ensemble est comme une carte avec seulement des clés, plutôt que des paires clé-valeur. En fait, les deux types d'ensembles de Rust, et , sont implémentés sous forme d'enveloppes minces autour de et . `HashSet<T> BTreeSet<T> HashMap<T, ()> BTreeMap<T, ()>`

`HashSet::new()`, `BTreeSet::new()`

Créez de nouveaux ensembles.

`iter.collect()`

Peut être utilisé pour créer un nouvel ensemble à partir de n'importe quel itérateur. Si produit des valeurs plus d'une fois, les doublons sont supprimés. `iter`

`HashSet::with_capacity(n)`

Crée un vide avec de l'espace pour au moins des valeurs. `HashSet n`

`HashSet<T>` et ont toutes les méthodes de base en commun: `BTreeSet<T>`

`set.len()`

Renvoie le nombre de valeurs dans . `set`

`set.is_empty()`

Renvoie si l'ensemble ne contient aucun élément. `true`

`set.contains(&value)`

Renvoie si l'ensemble contient le fichier . `true value`

`set.insert(value)`

Ajoute un à l'ensemble. Renvoie si une valeur a été ajoutée, si elle était déjà membre de l'ensemble. `value true false`

`set.remove(&value)`

Supprime un de l'ensemble. Renvoie si une valeur a été supprimée, si elle n'était pas déjà membre de l'ensemble. `value true false`

`set.retain(test)`

Supprime tous les éléments qui ne réussissent pas le test donné. L'argument est une fonction ou une fermeture qui implémente . Pour chaque élément de , cet appel , et s'il retourne , l'élément est supprimé de l'ensemble et supprimé. `test FnMut(&T) -> bool set test(&value) false`

En dehors de la performance, c'est comme écrire :

```
set = set.into_iter().filter(test).collect();
```

Comme pour les cartes, les méthodes qui recherchent une valeur par référence sont génériques sur les types qui peuvent être empruntés à . Pour plus de détails, voir « Emprunter et empruntermut ». `T`

Définir l'itération

Il existe deux façons d'itérer sur les ensembles :

- L'itération par valeur (« ») produit les membres de l'ensemble (et consomme l'ensemble). `for v in set`
- L'itération par référence partagée (« ») produit des références partagées aux membres de l'ensemble. `for v in &set`

L'itération sur un ensemble par référence n'est pas prise en charge. Il n'y a aucun moyen d'obtenir une référence à une valeur stockée dans un ensemble. `mut mut`

`set.iter()`

Renvoie un itérateur sur les membres de par référence. `set`

`HashSet` les itérateurs, comme les itérateurs, produisent leurs valeurs dans un ordre arbitraire. les itérateurs produisent des valeurs dans l'ordre, comme un vecteur trié. `HashMap BTreeSet`

Lorsque des valeurs égales sont différentes

Les ensembles ont quelques méthodes étranges que vous devez utiliser uniquement si vous vous souciez des différences entre les valeurs « égales ».

De telles différences existent souvent. Deux valeurs identiques, par exemple, stockent leurs caractères à différents endroits en mémoire : `String`

```
let s1 = "hello".to_string();
let s2 = "hello".to_string();
println!("{:p}", &s1 as &str);  // 0x7f8b32060008
println!("{:p}", &s2 as &str);  // 0x7f8b32060010
```

Habituellement, nous ne nous en soucions pas.

Mais si jamais vous le faites, vous pouvez accéder aux valeurs réelles stockées dans un ensemble en utilisant les méthodes suivantes. Chacun renvoie un qui est s'il ne contient pas de valeur correspondante : `Option None set`

`set.get(&value)`

Renvoie une référence partagée au membre de qui est égal à , le cas échéant. Renvoie un fichier . `set value Option<&T>`

`set.take(&value)`

Comme , mais il renvoie la valeur supprimée, le cas échéant. Renvoie un fichier
. `set.remove(&value) Option<T>`

`set.replace(value)`

Comme , mais si contient déjà une valeur égale à , cela remplace et renvoie l'ancienne valeur. Renvoie un fichier
. `set.insert(value) set value Option<T>`

Opérations de l'ensemble

Jusqu'à présent, la plupart des méthodes d'ensemble que nous avons vues sont axées sur une seule valeur dans un seul ensemble. Les ensembles ont également des méthodes qui fonctionnent sur des ensembles entiers :

`set1.intersection(&set2)`

Renvoie un itérateur sur toutes les valeurs qui se trouvent dans les deux et . `set1 set2`

Par exemple, si nous voulons imprimer les noms de tous les étudiants qui suivent à la fois des cours de chirurgie cérébrale et de science des fusées, nous pourrions écrire:

```
for student in &brain_class {
    if rocket_class.contains(student) {
        println!("{}", student);
    }
}
```

Ou, plus court :

```
for student in brain_class.intersection(&rocket_class) {
    println!("{}", student);
}
```

Étonnamment, il y a un opérateur pour cela.

`&set1 & &set2` renvoie un nouvel ensemble qui est l'intersection de et . Il s'agit de l'opérateur binaire binaire ET, appliqué à deux références. Cela trouve des valeurs qui sont dans les deux *et* : `set1 set2 set1 set2`

```
let overachievers = &brain_class & &rocket_class;
```

`set1.union(&set2)`

Renvoie un itérateur sur les valeurs qui se trouvent dans l'un ou l'autre ou , ou les deux. `set1 set2`

`&set1 | &set2` renvoie un nouvel ensemble contenant toutes ces valeurs. Il recherche les valeurs qui se trouvent dans *ou* . `set1 set2`

`set1.difference(&set2)`

Renvoie un itérateur sur les valeurs qui se trouvent dans mais pas dans . `set1 set2`

`&set1 - &set2` renvoie un nouvel ensemble contenant toutes ces valeurs.

`set1.symmetric_difference(&set2)`

Renvoie un itérateur sur les valeurs qui se trouvent dans l'un ou l'autre ou , mais pas les deux. `set1 set2`

`&set1 ^ &set2` renvoie un nouvel ensemble contenant toutes ces valeurs.

Et il existe trois méthodes pour tester les relations entre les ensembles :

`set1.is_disjoint(set2)`

True si et n'ont pas de valeurs en commun : l'intersection entre elles est vide. `set1 set2`

`set1.is_subset(set2)`

True if est un sous-ensemble de —c'est-à-dire que toutes les valeurs dans sont également dans . `set1 set2 set1 set2`

`set1.is_superset(set2)`

C'est l'inverse : c'est vrai si est un surensemble de . `set1 set2`

Les ensembles prennent également en charge les tests d'égalité avec et ; deux ensembles sont égaux s'ils contiennent les mêmes valeurs. `== !=`

Hachage

`std::hash::Hash` est le trait de bibliothèque standard pour les types hashables. les clés et les éléments doivent implémenter à la fois et . `HashMap HashSet Hash Eq`

La plupart des types intégrés qui implémentent implémentent également . Les types entiers, , et sont tous hashables; il en va de même pour les tu-

ples, les tableaux, les tranches et les vecteurs, tant que leurs éléments sont hachables. `Eq Hash char String`

L'un des principes de la bibliothèque standard est qu'une valeur doit avoir le même code de hachage, quel que soit l'endroit où vous la stockez ou la façon dont vous la pointez. Par conséquent, une référence a le même code de hachage que la valeur à laquelle elle fait référence et a le même code de hachage que la valeur encadrée. Un vecteur a le même code de hachage que la tranche contenant toutes ses données, . A a le même code de hachage que a avec les mêmes caractères. `Box vec &vec[..] String &str`

Les structs et les enums ne sont pas implémentés par défaut, mais une implémentation peut être dérivée : `Hash`

```
/// The ID number for an object in the British Museum's collection.
#[derive(Clone, PartialEq, Eq, Hash)]
enum MuseumNumber {
    ...
}
```

Cela fonctionne tant que les champs du type sont tous hachissables.

Si vous implémentez à la main pour un type, vous devez également implémenter à la main. Par exemple, supposons que nous ayons un type qui représente des trésors historiques inestimables : `PartialEq Hash`

```
struct Artifact {
    id: MuseumNumber,
    name: String,
    cultures: Vec<Culture>,
    date: RoughTime,
    ...
}
```

Deux s sont considérés comme égaux s'ils ont le même ID : `Artifact`

```
impl PartialEq for Artifact {
    fn eq(&self, other: &Artifact) -> bool {
        self.id == other.id
    }
}

impl Eq for Artifact {}
```

Puisque nous comparons les artefacts uniquement sur la base de leur ID, nous devons les hacher de la même manière:

```
use std::hash::{Hash, Hasher};

impl Hash for Artifact {
    fn hash<H: Hasher>(&self, hasher: &mut H) {
        // Delegate hashing to the MuseumNumber.
        self.id.hash(hasher);
    }
}
```

(Sinon, ne fonctionnerait pas correctement; comme toutes les tables de hachage, il nécessite que si .) `HashSet<Artifact> hash(a) == hash(b) a == b`

Cela nous permet de créer un de : `HashSet Artifacts`

```
let mut collection = HashSet::<Artifact>::new();
```

Comme le montre ce code, même lorsque vous implémentez à la main, vous n'avez pas besoin de savoir quoi que ce soit sur les algorithmes de hachage. reçoit une référence à un , qui représente l'algorithme de hachage. Il vous suffit d'alimenter toutes les données pertinentes pour l'opérateur. Le calcule un code de hachage à partir de tout ce que vous lui donnez. `Hash .hash() Hasher Hasher == Hasher`

Utilisation d'un algorithme de hachage personnalisé

La méthode est générique, de sorte que les implémentations montrées précédemment peuvent alimenter les données à n'importe quel type qui implémente . C'est ainsi que Rust prend en charge les algorithmes de hachage enfichables. `hash Hash Hasher`

Un troisième trait, , est le trait pour les types qui représentent l'état initial d'un algorithme de hachage. Chacun est à usage unique, comme un itérateur : vous l'utilisez une fois et vous le jetez. A est réutilisable. `std::hash::BuildHasher Hasher BuildHasher`

Chaque contient un qu'il utilise chaque fois qu'il a besoin de calculer un code de hachage. La valeur contient la clé, l'état initial ou d'autres

paramètres dont l'algorithme de hachage a besoin chaque fois qu'il s'exécute. `HashMap BuildHasher BuildHasher`

Le protocole complet pour le calcul d'un code de hachage ressemble à ceci :

```
use std::hash::{Hash, Hasher, BuildHasher};

fn compute_hash<B, T>(builder: &B, value: &T) -> u64
    where B: BuildHasher, T: Hash
{
    let mut hasher = builder.build_hasher();  // 1. start the algorithm
    value.hash(&mut hasher);                  // 2. feed it data
    hasher.finish()                           // 3. finish, producing a
}
```

`HashMap` appelle ces trois méthodes chaque fois qu'il doit calculer un code de hachage. Toutes les méthodes sont ininflammables, donc c'est très rapide.

L'algorithme de hachage par défaut de Rust est un algorithme bien connu appelé SipHash-1-3. SipHash est rapide et très bon pour minimiser les collisions de hachage. En fait, il s'agit d'un algorithme cryptographique : il n'existe aucun moyen efficace connu de générer des collisions SipHash-1-3. Tant qu'une clé différente et imprévisible est utilisée pour chaque table de hachage, Rust est protégé contre une sorte d'attaque par déni de service appelée HashDoS, où les attaquants utilisent délibérément des collisions de hachage pour déclencher les pires performances dans le pire des cas sur un serveur.

Mais peut-être que vous n'en avez pas besoin pour votre application. Si vous stockez de nombreuses petites clés, telles que des entiers ou des chaînes très courtes, il est possible d'implémenter une fonction de hachage plus rapide, au détriment de la sécurité HashDoS. La caisse implémente l'un de ces algorithmes, le hachage Fowler-Noll-Vo (FNV). Pour l'essayer, ajoutez cette ligne à votre *Cargo.toml*: `fnv`

```
[dependencies]
fnv = "1.0"
```

Importez ensuite la carte et définissez les types à partir de : `fnv`

```
use fnv::{FnvHashMap, FnvHashSet};
```

Vous pouvez utiliser ces deux types comme remplacements sans rendez-vous pour et . Un coup d'œil à l'intérieur du code source révèle comment ils sont définis : `HashMap HashSet fnv`

```
/// A `HashMap` using a default FNV hasher.
pub type FnvHashMap<K, V> = HashMap<K, V, FnvBuildHasher>;

/// A `HashSet` using a default FNV hasher.
pub type FnvHashSet<T> = HashSet<T, FnvBuildHasher>;
```

La norme et les collections acceptent un paramètre de type supplémentaire facultatif spécifiant l'algorithme de hachage ; et sont des alias de type générique pour et , spécifiant un hasher FNV pour ce paramètre. `HashMap HashSet FnvHashMap FnvHashSet HashMap Hash Set`

Au-delà des collections standard

La création d'un nouveau type de collection personnalisé dans Rust est à peu près la même que dans n'importe quelle autre langue. Vous organisez les données en combinant les parties fournies par le langage : structs et enums, collections standard, s, es, etc. Pour obtenir un exemple, voir le type défini dans « Generic Enums ». `Option Box BinaryTree<T>`

Si vous avez l'habitude d'implémenter des structures de données en C++, en utilisant des pointeurs bruts, la gestion manuelle de la mémoire, le placement et les appels de destructeur explicites pour obtenir les meilleures performances possibles, vous trouverez sans aucun doute Rust sûr plutôt limitatif. Tous ces outils sont intrinsèquement dangereux. Ils sont disponibles dans Rust, mais seulement si vous optez pour un code dangereux. Le chapitre 22 montre comment; il inclut un exemple qui utilise du code non sécurisé pour implémenter une collection personnalisée sécurisée. `new`

Pour l'instant, nous allons simplement nous prélasser dans la lueur chaleureuse des collections standard et de leurs API sûres et efficaces. Comme la plupart des bibliothèques standard Rust, ils sont conçus pour s'assurer que le besoin d'écrire est aussi rare que possible. `unsafe`