Chapitre 16. Collectes

> *Nous nous comportons tous comme le démon de Maxwell. Les organismes s'organisent. C'est dans l'expérience quotidienne que réside la raison pour laquelle des physiciens sobres au cours de deux siècles ont maintenu en vie ce fantasme de bande dessinée. Nous trions le courrier, construisons des châteaux de sable, résolvons des puzzles, séparons le blé de l'ivraie, réarrangeons les pièces d'échecs, collectionnons les timbres, classons les livres par ordre alphabétique, créons des symétries, composons des sonnets et des sonates et mettons de l'ordre dans nos pièces, et tout cela ne nécessite aucune 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*, types génériques pour stocker des données en mémoire. Nous avons déjà utilisé des collections, telles que `Vec` et `HashMap`, tout au long de ce livre. Dans ce chapitre, nous couvrirons en détail les méthodes de ces deux types, ainsi que les autres collections standard d'une demi-douzaine. Avant de commencer, abordons quelques différences systématiques entre les collections de Rust et celles d'autres langues.

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

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

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

En dehors de ces différences, les collections de Rust correspondent à ce que vous attendez. Si vous êtes un programmeur expérimenté pressé, vous pouvez survoler ici, mais ne manquez pas "Entries".

Aperçu

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

Tableau 16-1. Résumé des collections standards

Le recueil	La description	Type de collection similaire dans…		
		C++	Java	Python
`Vec<T>`	Tableau évolutif	`vector`	`ArrayList`	`list`
`VecDeque<T>`	File d'attente double (tampon circulaire extensible)	`deque`	`ArrayDeque`	`collections.deque`
`LinkedList<T>`	Liste doublement liée	`list`	`LinkedList`	—
`BinaryHeap<T>` where `T: Ord`	Tas maximum	`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`	Tableau des valeurs-clés triées	`map`	`TreeMap`	—
`HashSet<T>` where `T: Eq + Hash`	Ensemble non ordonné basé sur le hachage	`unordered_set`	`HashSet`	`set`

Le recueil	La description	Type de collection similaire dans…		
		C++	Java	Python
`BTreeSet<T>` où `T: Ord`	Ensemble trié	`set`	`TreeSet`	—

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

`Vec<T>`

Un cultivable, tableau de valeurs alloué par tas de type `T`. Environ la moitié de ce chapitre est consacrée à `Vec` ses nombreuses méthodes utiles.

`VecDeque<T>`

Comme `Vec<T>`, mais en mieuxà utiliser comme file d'attente premier entré, premier sorti. Il prend en charge efficacement l'ajout et la suppression de valeurs au début de la liste ainsi qu'à l'arrière. Cela se fait au prix de rendre toutes les autres opérations légèrement plus lentes.

`BinaryHeap<T>`

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

`HashMap<K, V>`

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

`HashSet<T>`

Un ensemble de valeurs de type `T`. 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.

`BTreeSet<T>`

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

Parce `LinkedList` 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.

Vec<T>

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

La façon la plus simple de créer un vecteur est d'utiliser la `vec!` macro:

```
// 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 a 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, `numbers` , a initialement une capacité de 0. Aucune mémoire de tas ne lui est allouée jusqu'à ce que le premier élément soit ajouté.

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

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

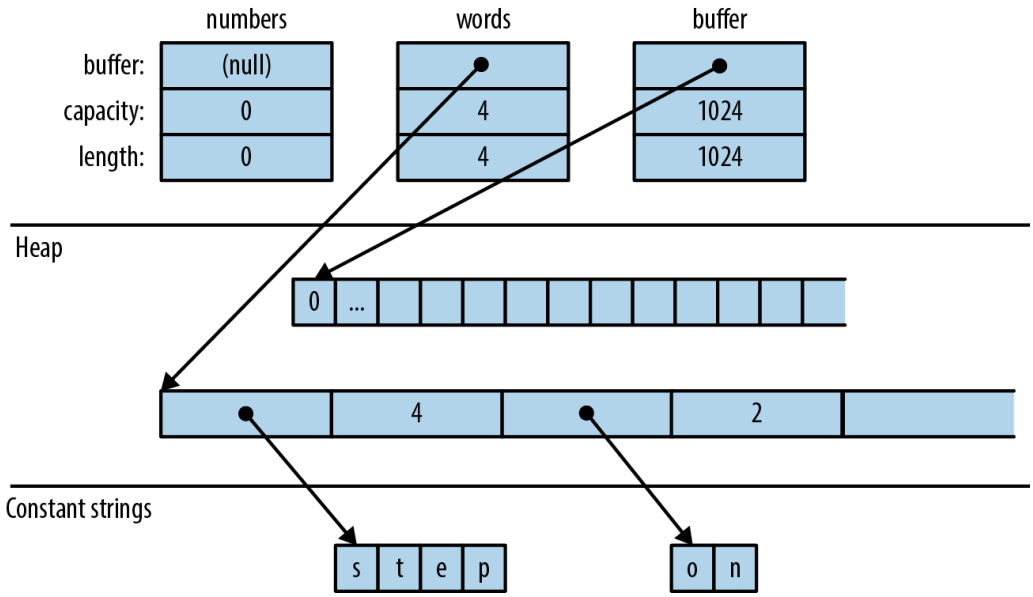

Illustration 16-1. Disposition vectorielle en mémoire : chaque élément de mots est une `&str` valeur composée d'un pointeur et d'une longueur

Accéder aux éléments

Obtenir des élémentsd'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 ne fait aucune exception pour les vecteurs. Longueurs et indices des vecteurssont de type `usize`. Essayer d'utiliser un `u32`, `u64` ou `isize` comme index vectoriel est une erreur. Vous pouvez utiliser un `n as usize` casting pour convertir au besoin; voir "Type Casts".

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()
```

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

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

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

`slice.last()`

Similairemais renvoie une référence au dernier élément.

`slice.get(index)`

Retour `Some` référence à `slice[index]`, si elle existe. Si `slice` a moins de `index+1` éléments, cela retourne `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)`

Variantesdes précédents qui empruntent `mut` les références :

```
let mut slice = [0, 1, 2, 3];
{
    let last = slice.last_mut().unwrap();    // type of last: &mut i32
    assert_eq!(*last, 3);
    *last = 100;
}
assert_eq!(slice, [0, 1, 2, 100]);
```

Étant donné que renvoyer une `T` valeur par signifierait la déplacer, les méthodes qui accèdent aux éléments en place renvoient généralement ces éléments par référence.

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

`slice.to_vec()`

Clonerune 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, ceest, `where T: Clone`.

Itération

Vecteurs, les tableaux et les tranches sont itérables, soit par valeur, soit par référence, en suivant le modèle décrit dans « IntoIterator Implementations » :

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

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

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

Vecteurs croissants et rétrécissants

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

`slice.len()`

Retourune `slice` longueur de , comme un `usize`.

`slice.is_empty()`

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

Les autres méthodes de cette section concernent les vecteurs croissants et rétrécissants. 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, alloué par tas. La *capacité*d'un vecteur est le nombre maximum d'éléments qui tiendraient dans ce morceau. `Vec` 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 de gestion explicite de la capacité :

`Vec::with_capacity(n)`

Créeun nouveau vecteur vide de capacité `n` .

`vec.capacity()`

`vec` Capacité de retour, en tant que `usize` . C'est toujours vrai que `vec.capacity() >= vec.len()` .

`vec.reserve(n)`

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

`vec.reserve_exact(n)`

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

`vec.shrink_to_fit()`

Essaiepour libérer de la mémoire supplémentaire si `vec.capacity()` est supérieur à `vec.len()` .

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

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

`vec.push(value)`

Ajoutele donné `value` à la fin de `vec` .

`vec.pop()`

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

Notez que `.push()` prend son argument par valeur, pas par référence. De même, `.pop()` renvoie la valeur sautée, pas une référence. Il en va de même pour la plupart des autres méthodes de cette section. Ils déplacent des valeurs dans et hors des vecteurs.

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

`vec.insert(index, value)`

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

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

`vec.remove(index)`

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

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

Plus le vecteur est long, plus l'opération est lente. Si vous vous retrouvez à en faire `vec.remove(0)` beaucoup, pensez à utiliser a `VecDeque` (expliqué dans « VecDeque<T> ») au lieu de a `Vec`.

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

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

`vec.resize(new_len, value)`

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

`vec.resize_with(new_len, closure)`

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

`vec.truncate(new_len)`

Réduitla longueur de `vec` à `new_len` , supprimant tous les éléments qui se trouvaient dans la plage `vec[new_len..]` .

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

`vec.clear()`

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

Quatre méthodes ajoutent ou suppriment plusieurs valeurs à la fois :

`vec.extend(iterable)`

Ajoutetous les éléments à partir de la `iterable` valeur donnée à la fin de `vec` , dans l'ordre. C'est comme une version multivaleur de `.push()` . L' `iterable` argument peut être tout ce qui implémente `IntoIterator<Item=T>` .

Cette méthode est si utile qu'il existe un trait standard pour cela, le `Extend` trait, que toutes les collections standard implémentent. Malheureusement, cela provoque `rustdoc` un regroupement `.extend()` avec d'autres méthodes de trait dans une grosse 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 "Le trait d'extension" pour plus d'informations.

`vec.split_off(index)`

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

`vec.append(&mut vec2)`

Cettedéplace tous les éléments de `vec2` vers `vec` , où `vec2` est un autre vecteur de type `Vec<T>` . Après, `vec2` c'est vide.

C'est comme `vec.extend(vec2)` sauf qu'il `vec2` existe toujours après, avec sa capacité non affectée.

`vec.drain(range)`

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

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

vec.retain(test)

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

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

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

vec.dedup()

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

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

Notez qu'il y a encore deux `'s'` 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 `.dedup()` , déplacer les données dans un ensemble ou (pour conserver les éléments dans leur ordre d'origine) utiliser cette `.retain()` astuce :

```
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 `.insert()` revient `false` lorsque l'ensemble contient déjà l'élément que nous insérons.

vec.dedup_by(same)

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

`vec.dedup_by_key(key)`

Le mêmecomme `vec.dedup()`, mais il traite deux éléments comme égaux si `key(&mut elem1) == key(&mut elem2)`.

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

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

De toutes les méthodes couvertes dans cette section, seules `.resize()` les valeurs sont clonées. Les autres fonctionnent en déplaçant des valeurs d'un endroit à un autre.

Joindre

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

`slices.concat()`

Retourun 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)`

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

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

Scission

C'est facile d'en avoir plusieursdes non `mut` -références dans un tableau, une tranche ou un vecteur à la fois :

```
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 `mut` références n'est pas si simple :

```
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 car if `i == j` , then `a` et `b` seraient deux `mut` références au même entier, en violation des règles de sécurité de Rust. (Voir "Partage contre mutation" .)

Rust a plusieurs méthodes qui peuvent emprunter `mut` 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 `mut` -slices, il existe donc des `mut` non-`mut` versions de chacune.

La figure 16-2 illustre ces méthodes.

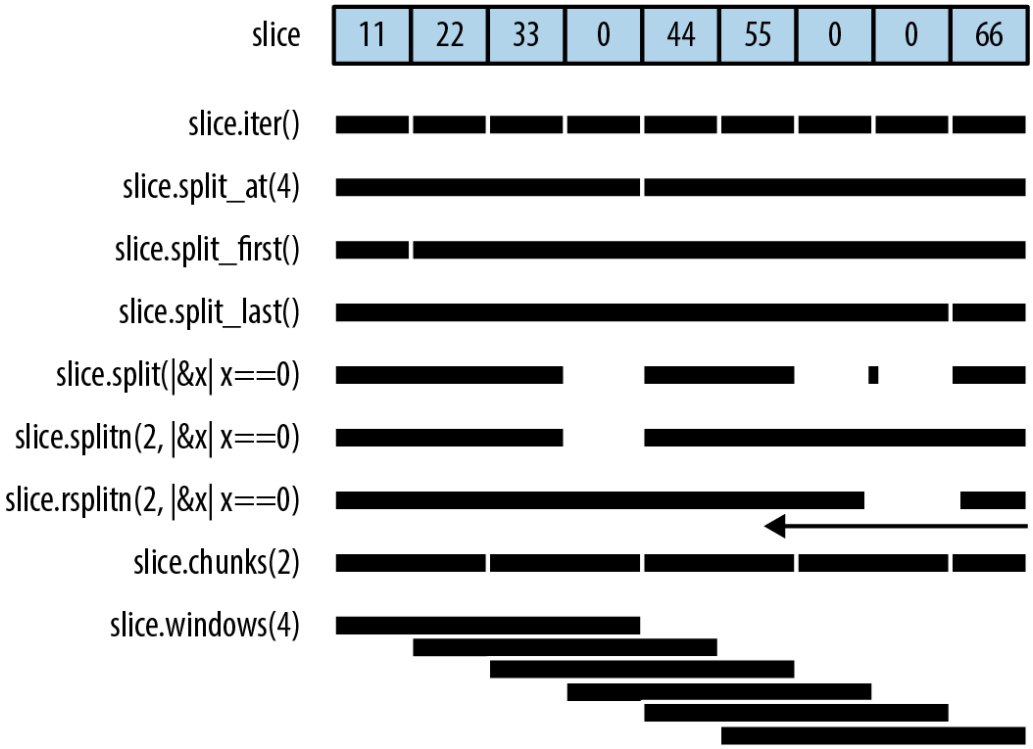

Illustration 16-2. Méthodes de fractionnement illustrées (remarque : le petit rectangle dans la sortie de `slice.split()` est une tranche vide causée par les deux séparateurs adjacents, et `rsplitn` produit sa sortie dans l'ordre de bout en bout, contrairement aux autres)

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()`

Produireune référence à chaque élément de `slice`. Nous les avons couverts dans "Itération".

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

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

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

Aussirenvoie une paire : une référence au premier élément (`slice[0]`) et une référence de tranche à tous les autres (`slice[1..]`).

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

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

Cessont analogues mais séparent le dernier élément plutôt que le premier.

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

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

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

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

La sortie contient toujours au moins une sous-tranche, plus une par séparateur. Les sous-tranches 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 `split` et `split_mut`, mais incluent le séparateur à la fin de la sous-tranche précédente plutôt que de l'exclure.

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

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

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

Le mêmemais ils produisent au plus des `n` sous-tranches. Une fois les premières `n-1` tranches trouvées, `is_sep` n'est plus appelée. La dernière sous-tranche contient tous les éléments restants.

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

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

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

Revenirun itérateur sur des sous-tranches non superposées de longueur `n`. Si `n` ne se divise pas `slice.len()` exactement, le dernier morceau contiendra moins de `n` éléments.

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

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

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

Revenirun itérateur sur des sous-tranches non superposées de longueur `n`. Si `n` ne divise pas `slice.len()`, le dernier morceau (avec moins de `n` éléments) est disponible dans la `remainder()` méthode du résultat.

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

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

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

`slice.windows(n)`

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

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

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

Une fenêtre glissante 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' temps
                  .map(|w| w[1] - w[0])    // how much did it change?
                  .collect::<Vec<_>>();
```

Parce que les sous-tranches se chevauchent, il n'y a pas de variation de cette méthode qui renvoie `mut` des références.

Échange

Làsont des méthodes pratiques pour échanger le contenu des tranches :

`slice.swap(i, j)`

Échangesles deux éléments `slice[i]` et `slice[j]`.

`slice_a.swap(&mut slice_b)`

Permute tout le contenu de `slice_a` et `slice_b` . `slice_a` et `slice_b` doit être de la même longueur.

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

`vec.swap_remove(i)`

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

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

Tri et recherche

Tranchesproposent trois méthodes de tri :

`slice.sort()`

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

`slice.sort_by(cmp)`

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

La mise en œuvre manuelle `cmp` est pénible, à moins que vous ne déléguiez à une `.cmp()` méthode :

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

Pour trier par un champ, en utilisant un deuxième champ comme condition de départage, 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)`

Trieles éléments de `slice` dans un ordre croissant par une clé de tri, donnée par la fonction ou la fermeture `key`. Le type de `key` doit implémenter `Fn(&T) -> K` où `K: Ord`.

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

```
// 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 lors du tri, de sorte que la `key` fonction peut être appelée plus de *n* fois.

Pour des raisons techniques, `key(element)` ne peut renvoyer aucune référence empruntée à l'élément. Cela ne fonctionnera pas :

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

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

`slice.reverse()`

Reversune 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 recherchespour `value` dans le trié donné `slice`. Remarque qui `value` est passé par référence.

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

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

Puisque `f32` et `f64` ont des valeurs NaN, ils ne s'implémentent pas `Ord` 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 `ord_subset` crate.

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

slice.contains(&value)

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

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

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

Cela renvoie un `Option<usize>`.

Comparer des tranches

Si un type prend en `T` chargeles opérateurs `==` et (le trait, décrit dans « Comparaisons d'équivalence »), puis les tableaux , les tranches et les vecteurs les prennent également en charge. Deux tranches sont égales si elles ont la même longueur et leurs éléments correspondants sont égaux.

Il en va de même pour les tableaux et les vecteurs. `!= PartialEq [T; N] [T] Vec<T>`

Si `T` prend en charge les opérateurs `<`, `<=`, `>`, et `>=` (le `PartialOrd` trait, décrit dans « Comparaisons ordonnées »), alors les tableaux, les tranches et les vecteurs de `T` do aussi. Les comparaisons de tranches sont lexicographiques.

Deux méthodes pratiques effectuent des comparaisons de tranches courantes :

`slice.starts_with(other)`

Retour `true` if `slice` commence par une séquence de valeurs égales aux éléments de la 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)`

Similairemais vérifie la fin de `slice` :

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

Éléments aléatoires

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

`slice.choose(&mut rng)`

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

`slice.shuffle(&mut rng)`

Au hasardréordonne les éléments d'une tranche en place. La tranche doit être passée par `mut` référence.

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

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

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

Rust élimine les erreurs d'invalidation

Le plus grand publicles langages de programmation ont des collections et des itérateurs, et ils ont tous une certaine variation sur 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 essayons 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 `enumerate` fonction est l'équivalent Python de la `.enumerate()` méthode de Rust, décrite dans "enumerate" .)

Ce programme, étonnamment, imprime `[1, 3, 7]`. Mais sept est plus grand que quatre. Comment cela s'est-il passé? 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++ , c'est un comportement indéfini. En Python, bien que le comportement soit bien défini, il n'est pas intuitif : l'itérateur ignore un élément. `val` n'est jamais `7` .

Essayons de reproduire ce bogue 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 muta
        }
    }
```

```
    println!("{:?}", my_vec);
}
```

Naturellement, Rust rejette ce programme au moment de la compilation. Lorsque nous appelons `my_vec.iter()`, il emprunte une référence partagée (non- `mut`) au vecteur. La référence vit aussi longtemps que l'itérateur, jusqu'à la fin de la `for` boucle. Nous ne pouvons pas modifier le vecteur en appelant `my_vec.remove(index)` alors qu'une non- `mut` référence existe.

Se faire signaler une erreur, c'est bien, mais bien sûr, encore faut-il 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 tout autre langage : créer un nouveau vecteur en utilisant un `filter`.

VecDeque<T>

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

Rust's `std::collections::VecDeque<T>` 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 :

`deque.push_front(value)`

Ajouteune valeur au début de la file d'attente.

`deque.push_back(value)`

Ajouteune valeur à la fin. (Cette méthode est beaucoup plus utilisée que `.push_front()`, 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.)

`deque.pop_front()`

Supprimeet renvoie la valeur avant de la file d'attente, renvoyant un `Option<T>` c'est-à-dire `None` si la file d'attente est vide, comme `vec.pop()`.

`deque.pop_back()`

Supprimeet renvoie la valeur à l'arrière, renvoyant à nouveau un `Option<T>` .

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

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

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

Travaillercomme `vec.first_mut()` et `vec.last_mut()` , revenant `Option<&mut T>` .

L'implémentation de `VecDeque` est une mémoire tampon en anneau, comme illustré à la Figure 16-3 .

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

Ajouter une valeur à la file d'attente, à chaque extrémité, signifie revendiquer l'un des emplacements inutilisés, illustré par les blocs les plus sombres, boucler ou allouer une plus grande quantité de mémoire si nécessaire.

`VecDeque` gère l'emballage, vous n'avez donc pas à y penser. La figure 16-3 est une vue des coulisses de la `.pop_front()` rapidité de Rust.

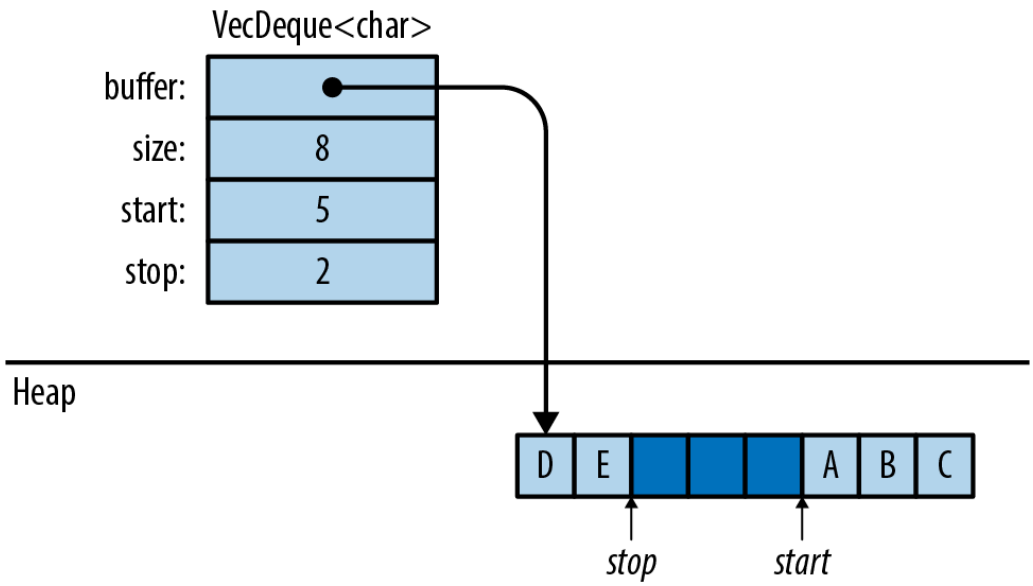

Illustration 16-3. Comment a `VecDeque` est stocké en mémoire

Souvent, lorsque vous avez besoin d'une deque, `.push_back()` et `.pop_front()` sont les deux seules méthodes dont vous aurez besoin. Les fonctions associées au type `VecDeque::new()` et `VecDeque::with_capacity(n)`, pour la création de files d'attente, sont identiques à leurs homologues dans `Vec`. De nombreuses `Vec` méthodes sont également implémentées pour `VecDeque` : `.len()` et `.is_empty()`, `.insert(index, value)`, `.remove(index)`, `.extend(iterable)`, etc.

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

Comme les deques ne stockent pas leurs éléments de manière contiguë en 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, `VecDeque` fournit une méthode qui résoudra cela :

`deque.make_contiguous()`

Prend `&mut self` et réorganisele `VecDeque` dans la mémoire contiguë, retournant `&mut [T]`.

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

`Vec::from(deque)`

`Vec<T>` met en œuvre `From<VecDeque<T>>`, donc cela tourne une dequedans un vecteur. Cela coûte O(*n*) en temps, car cela peut nécessiter un réarrangement des éléments.

`VecDeque::from(vec)`

`VecDeque<T>` met en œuvre `From<Vec<T>>`, donc cela transforme un vecteurdans une 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 le nouveau deque.

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

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

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

BinaryHeap<T>

A `BinaryHeap` est une collectiondont les éléments sont organisés de manière lâche afin que la plus grande valeur bouillonne toujours au début de la file d'attente. Voici les trois `BinaryHeap` méthodes les plus couramment utilisées :

`heap.push(value)`

Ajouteune valeur au tas.

`heap.pop()`

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

`heap.peek()`

Retourune référence à la plus grande valeur du tas. Le type de retour est `Option<&T>` .

`heap.peek_mut()`

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

```
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 `Vec` , y compris `BinaryHeap::new()` , `.len()` , `.is_empty()` , `.capacity()` , `.clear()` et `.append(&mut heap2)` .

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

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

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

La valeur `9` est en haut du tas :

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

La suppression de la valeur `9` réorganise également légèrement les autres éléments pour qu'ils `8` soient maintenant au premier plan, et ainsi de suite :

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

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

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

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

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

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

Une *carte* est une collectionde paires clé-valeur (appelées *entrées*). Deux entrées n'ont pas la même clé et les entrées sont organisées de sorte que si vous avez une clé, vous pouvez rechercher efficacement la valeur cor-

respondante dans une carte. En bref, une carte est une table de recherche.

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

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

La figure 16-4 montre comment a `HashMap` 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 par tas. L'ajout d'entrées force éventuellement le `HashMap` à allouer une table plus grande et à y déplacer toutes les données.

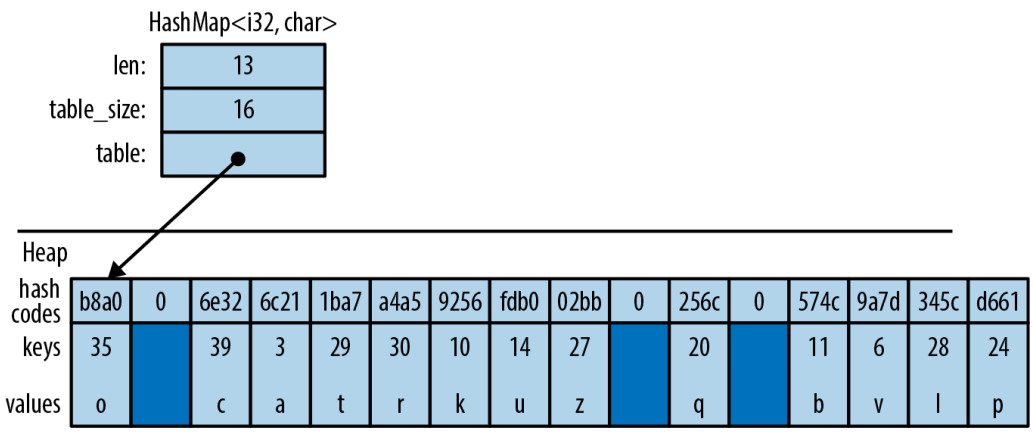

Illustration 16-4. A `HashMap` en mémoire

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

A `BTreeMap` stocke ses entrées dans des *nœuds* . La plupart des nœuds d'un `BTreeMap` 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. Le pointeur entre `(20, 'q')` et `(30, 'r')` pointe vers un nœud enfant contenant des clés entre `20` et `30` . 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.

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

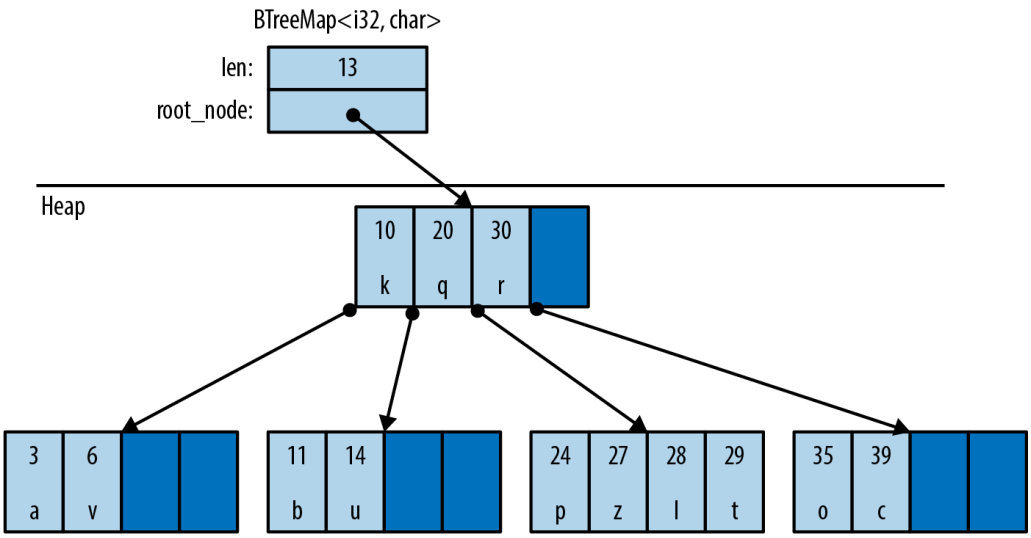

Illustration 16-5. A `BTreeMap` en mémoire

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 mémoire sont regroupés plutôt que dispersés sur l'ensemble du tas. Cela rend les échecs du cache CPU plus rares. C'est un gain de vitesse significatif.

Il existe plusieurs façons de créer une carte :

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

Créez de nouvelles cartes vides.

`iter.collect()`

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

`HashMap::with_capacity(n)`

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

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

`map.len()`

Retourle nombre d'entrées.

`map.is_empty()`

Retour `true` si `map` n'a pas d'entrées.

`map.contains_key(&key)`

Retour `true` si la carte a une entrée pour le donné `key` .

`map.get(&key)`

Recherche `map` une entréeavec le donné `key` . Si une entrée correspondante est trouvée, cela renvoie `Some(r)` , où `r` est une référence à la valeur correspondante. Sinon, cela renvoie `None` .

`map.get_mut(&key)`

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

En général, les cartes vous permettent d' `mut` accéder aux valeurs stockées à l'intérieur, mais pas aux clés. Les valeurs sont à vous de modifier comme bon vous semble. 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 sur place serait un bogue.

`map.insert(key, value)`

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

`map.extend(iterable)`

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

`map.append(&mut map2)`

Se déplacetoutes les entrées de `map2` dans `map` . Après, `map2` c'est vide.

`map.remove(&key)`

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

`map.remove_entry(&key)`

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

`map.retain(test)`

Supprimetous les éléments qui ne passent pas le test donné. L' `test` argument est une fonction ou une fermeture qui implémente `FnMut(&K, &mut V) -> bool` . Pour chaque élément de `map` ,

cela appelle `test(&key, &mut value)`, et s'il renvoie `false`, l'élément est supprimé de la carte et supprimé.

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

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

`map.clear()`

Supprimetoutes les entrées.

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

L' `key` argumentationà `.contains_key()`, `.get()`, `.get_mut()`, et `.remove()` n'a pas besoin d'avoir le type exact `&K`. Ces méthodes sont génériques par rapport aux types qui peuvent être empruntés à `K`. Il est acceptable d'appeler `fish_map.contains_key("conger")` un `HashMap<String, Fish>`, même s'il `"conger"` ne s'agit pas exactement d'un `String`, car `String` implements `Borrow<&str>`. Pour plus de détails, voir "Emprunter et EmprunterMut".

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

`btree_map.split_off(&key)`

Se divise `btree_map` en deux. Les entrées avec des clés inférieures à `key` sont laissées dans `btree_map`. Retourne un new `BTreeMap<K, V>` contenant les autres entrées.

Entrées

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

```
// 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 `student_map` deux ou trois fois, en faisant la même recherche à chaque fois.

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

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

La `Entry` valeur renvoyée par `student_map.entry(name.to_string())` agit comme une référence mutable à un emplacement 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 à cet endroit. S'il est vacant, la `.or_insert_with()` méthode de l'entrée insère un nouveau `Student` . La plupart des utilisations des entrées sont comme ceci : courtes et douces.

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

`map.entry(key)`

Retourun `Entry` pour le donné `key` . S'il n'y a pas une telle clé dans la carte, cela renvoie un vacant `Entry` .

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

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

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

De retour dans "Structs Containing References" , nous avons vu comment stocker des références dans un type et comment cela affecte les durées de vie. Nous voyons maintenant 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 `&str` à cette méthode si la carte a des `String` clés. La `.entry()` méthode, dans ce cas, nécessite un réel `String`.

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

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

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

Supposons que nous ayons besoin de 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 `mut` référence, donc le type de `count` est `&mut usize`.

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

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

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

Cetteest identique, sauf que s'il doit créer une nouvelle entrée, il appelle `default_fn()` pour produire la valeur par défaut. S'il existe déjà une entrée pour `key` dans le `map`, then `default_fn` n'est pas utilisé.

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 `closure` si une entréeavec la clé `key` existe, en passant une référence mutable à la valeur. Il renvoie le `Entry` , il peut donc être enchaîné avec d'autres méthodes.

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 `Entry` type est une énumération, définie comme ceci pour `HashMap` (et de la même manière pour `BTreeMap`):

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

Les types `OccupiedEntry` et `VacantEntry` 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 `.or_insert()` et `.or_insert_with()` couvrir les cas courants.

Itération de carte

Il existe plusieurs façons d'itérersur une carte :

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

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

`map.keys()`

Retourun itérateur sur les clés seulement, par référence.

`map.values()`

Retourun itérateur sur les valeurs, par référence.

`map.values_mut()`

Retourun itérateur sur les valeurs, par `mut` référence.

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

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

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

HashSet<T> et BTreeSet<T>

*Ensembles*sont des collectionsde valeurs organisées pour un test d'adhésion rapide :

```
let b1 = large_vector.contains(&"needle");    // slow, checks every elemen
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 uniquement des clés, plutôt que des paires clé-valeur. En fait, les deux types d'ensembles de

Rust, `HashSet<T>` et `BTreeSet<T>`, sont implémentés comme des enveloppes minces autour de `HashMap<T, ()>` et `BTreeMap<T, ()>`.

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

Créernouveaux ensembles.

`iter.collect()`

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

`HashSet::with_capacity(n)`

Créeun vide `HashSet` avec de la place pour au moins `n` des valeurs.

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

`set.len()`

Retourle nombre de valeurs dans `set`.

`set.is_empty()`

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

`set.contains(&value)`

Retour `true` si l'ensemble contient le donné `value`.

`set.insert(value)`

Ajoutea `value` à l'ensemble. Renvoie `true` si une valeur a été ajoutée, `false` si elle faisait déjà partie de l'ensemble.

`set.remove(&value)`

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

`set.retain(test)`

Supprimetous les éléments qui ne passent pas le test donné. L'`test` argument est une fonction ou une fermeture qui implémente `FnMut(&T) -> bool`. Pour chaque élément de `set`, cela appelle `test(&value)`, et s'il renvoie `false`, l'élément est supprimé de l'ensemble et supprimé.

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 par rapport aux types qui peuvent être emprun-

tés à `T`. Pour plus de détails, voir "Emprunter et EmprunterMut" .

Définir l'itération

Il y a deux façons d'itérersur ensembles :

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

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

`set.iter()`

Retourun itérateur sur les membres de `set` par référence.

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

Lorsque des valeurs égales sont différentes

Les ensembles ont quelques méthodes étrangesque vous devez utiliser uniquement si vous vous souciez des différences entre les valeurs "égales".

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

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

D'habitude, on s'en fout.

Mais au cas où vous le feriez, vous pouvez accéder aux valeurs réelles stockées dans un ensemble en utilisant les méthodes suivantes. Chacun renvoie une valeur `Option` si `None` elle `set` ne contient pas de valeur correspondante :

`set.get(&value)`

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

`set.take(&value)`

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

`set.replace(value)`

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

Opérations sur tout l'ensemble

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

`set1.intersection(&set2)`

Retourun itérateur sur toutes les valeurs qui sont à la fois dans `set1` et `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 `set1` et `set2`. Il s'agit de l'opérateur AND binaire au niveau du

bit, appliqué à deux références. Cela trouve les valeurs qui sont à la fois dans `set1` *et* `set2` :

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

`set1.union(&set2)`

Retourun itérateur sur les valeurs qui sont dans `set1` ou `set2`, ou les deux.

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

`set1.difference(&set2)`

Retourun itérateur sur les valeurs qui sont dans `set1` mais pas dans `set2`.

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

`set1.symmetric_difference(&set2)`

Retourun itérateur sur les valeurs qui sont dans `set1` ou `set2`, mais pas les deux.

`&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)`

Vraisi `set1` et `set2` n'ont pas de valeurs en commun — l'intersection entre elles est vide.

`set1.is_subset(set2)`

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

`set1.is_superset(set2)`

Cetteest l'inverse : c'est vrai si `set1` est un sur-ensemble de `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 hashableles types. `HashMap` les clés et `HashSet` les éléments doivent implémenter à la fois `Hash` et `Eq`.

La plupart des types intégrés quimettre en œuvre `Eq` également mettre en œuvre `Hash`. Les types entiers, `char`, et `String` sont tous hachables ; il en va de même pour les tuples, les tableaux, les tranches et les vecteurs, tant que leurs éléments sont hachables.

Un principe 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 manière dont vous la pointez. Par conséquent, une référence a le même code de hachage que la valeur à laquelle elle se réfère, et a `Box` a le même code de hachage que la valeur encadrée. Un vecteur `vec` a le même code de hachage que la tranche contenant toutes ses données, `&vec[..]`. A `String` a le même code de hachage que a `&str` avec les mêmes caractères.

Structureset énumérationsne pas implémenter `Hash` par défaut, mais une implémentation peut être dérivée :

```
/// 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 hachables.

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

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

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

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

impl Eq for Artifact {}
```

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

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

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

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

Utilisation d'un algorithme de hachage personnalisé

La `hash` méthodeest générique, de sorte que les `Hash` implémentations présentées précédemment peuvent fournir des données à tout type qui implémente `Hasher`. C'est ainsi que Rust prend en charge les algorithmes de hachage enfichables.

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

Chaque `HashMap` contient un `BuildHasher` qu'il utilise chaque fois qu'il a besoin de calculer un code de hachage. La `BuildHasher` valeur contient la clé, l'état initial ou d'autres paramètres dont l'algorithme de hachage a besoin à chaque exécution.

Le protocole complet pour calculer 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 u6
}
```

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

L'algorithme de hachage par défaut de Rust est un algorithme bien connu appeléSipHash-1-3. SipHash est rapide et très efficace 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 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 `fnv` caisseimplémente un tel algorithme, le hachage Fowler – Noll – Vo (FNV). Pour l'essayer, ajoutez cette ligne à votre *Cargo.toml* :

```
[dépendances]
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 en remplacement de `HashMap` et `HashSet` . Un coup d'œil dans le `fnv` code source révèle comment ils sont définis :

```
/// 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 `HashMap` et `HashSet` les collections acceptent un paramètre de type supplémentaire facultatif spécifiant l'algorithme de hachage ; `FnvHashMap` et `FnvHashSet` sont des alias de type génériques pour `HashMap` et `HashSet` , spécifiant un hachage FNV pour ce paramètre.

Au-delà des collections standard

Créer un nouveau, personnaliséLe type de collection dans Rust est à peu près le même que dans n'importe quel autre langage. Vous organisez les données en combinant les parties fournies par le langage : structures et énumérations, collections standard, `Option` s, `Box` es, etc. Pour un exemple, voir le `BinaryTree<T>` type défini dans "Generic Enums" .

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

Pour l'instant, nous nous contenterons de profiter de la lueur chaleureuse des collections standard et de leurs API sûres et efficaces. Comme une grande partie de la bibliothèque standard Rust, ils sont conçus pour garantir que le besoin d'écrire `unsafe` est aussi rare que possible.