Chapitre 13. Caractéristiques utilitaires

> *La science n'est rien d'autre que la recherche de l'unité dans la variété sauvage de la nature - ou, plus exactement, dans la variété de notre expérience. La poésie, la peinture, les arts sont la même recherche, selon l'expression de Coleridge, de l'unité dans la variété.*
>
> —Jacob Bronowski

Ce chapitredécrit ce que nous appelons les traits « utilitaires » de Rust, un ensemble de divers traits de la bibliothèque standard qui ont suffisamment d'impact sur la façon dont Rust est écrit pour que vous deviez vous familiariser avec eux afin d'écrire du code et de la conception idiomatiques des interfaces publiques pour vos caisses que les utilisateurs jugeront comme étant proprement « rustiques ». Ils se répartissent en trois grandes catégories :

Traits d'extension de langue

Tout comme l'opérateurles traits de surcharge que nous avons abordés dans le chapitre précédent vous permettent d'utiliser les opérateurs d'expression de Rust sur vos propres types, il existe plusieurs autres traits de bibliothèque standard qui servent de points d'extension Rust, vous permettant d'intégrer plus étroitement vos propres types avec le langage. Ceux-ci incluent `Drop`, `Deref` et `DerefMut`, et les traits de conversion `From` et `Into`. Nous allons les décrire dans ce chapitre.

Traits marqueurs

Ce sont des traitsprincipalement utilisé pour lier des variables de type générique pour exprimer des contraintes que vous ne pouvez pas capturer autrement. Ceux-ci incluent `Sized` et `Copy`.

Traits de vocabulaire public

Ceux-ci n'ont rien de magiqueintégration du compilateur ; vous pouvez définir des traits équivalents dans votre propre code. Mais ils servent l'objectif important d'établir des solutions conventionnelles pour des problèmes communs. Celles-ci sont particulièrement utiles dans les interfaces publiques entre les caisses et les modules : en réduisant les variations inutiles, elles facilitent la compréhension des interfaces, mais elles augmentent également la probabilité que les fonctionnalités de différentes caisses puissent simplement être connectées directement ensemble, sans passe-partout ou code de colle personnalisé. Ceux-ci incluent , `Default` les traits d'emprunt de référence `AsRef`, et `AsMut` ; les traits de conversion faillibles et ; et le trait, une

généralisation de

. `Borrow` `BorrowMut` `TryFrom` `TryInto` `ToOwned` `Clone`

Ceux-ci sont résumés dans le Tableau 13-1 .

Tableau 13-1. Résumé des traits d'utilité

Caractéristique	La description
`Drop`	Destructeurs. Code de nettoyage que Rust exécute automatiquement chaque fois qu'une valeur est supprimée.
`Sized`	Trait de marqueur pour les types avec une taille fixe connue au moment de la compilation, par opposition aux types (tels que les tranches) qui sont dimensionnés dynamiquement.
`Clone`	Types prenant en charge les valeurs de clonage.
`Copy`	Trait de marqueur pour les types qui peuvent être clonés simplement en faisant une copie octet par octet de la mémoire contenant la valeur.
`Deref` et `DerefMut`	Caractéristiques pour les types de pointeurs intelligents.
`Default`	Les types qui ont une "valeur par défaut" sensible.
`AsRef` et `AsMut`	Traits de conversion pour emprunter un type de référence à un autre.
`Borrow` et `BorrowMut`	Traits de conversion, comme `AsRef` / `AsMut`, mais garantissant en outre un hachage, un ordre et une égalité cohérents.
`From` et `Into`	Traits de conversion pour transformer un type de valeur en un autre.
`TryFrom` et `TryInto`	Traits de conversion pour transformer un type de valeur en un autre, pour les transformations qui pourraient échouer.
`ToOwned`	Trait de conversion pour convertir une référence en une valeur possédée.

Il existe également d'autres caractéristiques de bibliothèque standard importantes. Nous couvrirons `Iterator` et `IntoIterator` au chapitre 15 . Le `Hash` trait, pour le calcul des codes de hachage, est traité au chapitre 16 . Et une paire de traits qui marquent les types thread-safe, `Send` et `Sync` , sont traités dans le chapitre 19 .

Goutte

Lorsque le propriétaire d'une valeur s'en va, on dit que Rust *tombe*la valeur. La suppression d'une valeur implique la libération de toutes les autres valeurs, du stockage de tas et des ressources système que la valeur possède. Les baisses se produisent dans diverses circonstances : lorsqu'une variable sort de la portée ; à la fin d'une instruction d'expression ; lorsque vous tronquez un vecteur, supprimez des éléments de sa fin ; etc.

Pour la plupart, Rust gère automatiquement la suppression des valeurs pour vous. Par exemple, supposons que vous définissiez le type suivant :

```
struct Appellation {
    name: String,
    nicknames:Vec<String>
}
```

An `Appellation` possède un tas de stockage pour le contenu des chaînes et le tampon d'éléments du vecteur. Rust s'occupe de nettoyer tout cela chaque fois qu'un `Appellation` est tombé, sans aucun autre codage nécessaire de votre part. Cependant, si vous le souhaitez, vous pouvez personnaliser la manière dont Rust supprime les valeurs de votre type en implémentant le `std::ops::Drop` trait :

```
trait Drop {
    fn drop(&mut self);
}
```

Une implémentation de `Drop` est analogue à un destructeur en C++ ou à un finaliseur dans d'autres langages. Lorsqu'une valeur est supprimée, si elle implémente `std::ops::Drop` , Rust appelle sa `drop` méthode, avant de procéder à la suppression des valeurs propres à ses champs ou éléments, comme il le ferait normalement. Cette invocation implicite de `drop` est le seul moyen d'appeler cette méthode ; si vous essayez de l'invoquer explicitement vous-même, Rust le signale comme une erreur.

Étant donné que Rust appelle `Drop::drop` une valeur avant de supprimer ses champs ou éléments, la valeur que la méthode reçoit est toujours entièrement initialisée. Une implémentation de `Drop` pour notre `Appellation` type peut utiliser pleinement ses champs :

```
impl Drop for Appellation {
    fn drop(&mut self) {
        print!("Dropping {}", self.name);
        if !self.nicknames.is_empty() {
            print!(" (AKA {})", self.nicknames.join(", "));
        }
        println!("");
    }
}
```

Compte tenu de cette implémentation, nous pouvons écrire ce qui suit :

```
{
    let mut a = Appellation {
        name: "Zeus".to_string(),
        nicknames:vec!["cloud collector".to_string(),
                       "king of the gods".to_string()]
    };

    println!("before assignment");
    a = Appellation { name: "Hera".to_string(), nicknames:vec![] };
    println!("at end of block");
}
```

Lorsque nous affectons le second `Appellation` à `a` , le premier est supprimé, et lorsque nous quittons la portée de `a` , le second est supprimé. Ce code imprime ce qui suit :

```
before assignment
Dropping Zeus (AKA cloud collector, king of the gods)
at end of block
Dropping Hera
```

Puisque notre `std::ops::Drop` implémentation de `Appellation` ne fait rien d'autre qu'afficher un message, comment, exactement, sa mémoire est-elle nettoyée ? Le `Vec` type implémente `Drop` , supprimant chacun de ses éléments, puis libérant le tampon alloué par tas qu'ils occupaient. A `String` utilise a en `Vec<u8>` interne pour contenir son texte, il n'a donc `String` pas besoin de s'implémenter `Drop` lui-même ; il lui

laisse `Vec` le soin de libérer les personnages. Le même principe s'étend aux `Appellation` valeurs : lorsqu'une est supprimée, c'est finalement l'`Vec` implémentation de `Drop` qui s'occupe de libérer le contenu de chacune des chaînes, et enfin de libérer le tampon contenant les éléments du vecteur. Quant à la mémoire qui contient le `Appellation` valeur elle-même, elle a aussi un propriétaire, peut-être une variable locale ou une structure de données, qui est responsable de sa libération.

Si la valeur d'une variable est déplacée ailleurs, de sorte que la variable n'est pas initialisée lorsqu'elle sort de la portée, alors Rust n'essaiera pas de supprimer cette variable : il n'y a aucune valeur à supprimer.

Ce principe est valable même lorsqu'une variable peut ou non avoir vu sa valeur s'éloigner, selon le flux de contrôle. Dans de tels cas, Rust garde une trace de l'état de la variable avec un indicateur invisible indiquant si la valeur de la variable doit être supprimée ou non :

```
let p;
{
    let q = Appellation { name: "Cardamine hirsuta".to_string(),
                          nicknames:vec!["shotweed".to_string(),
                                         "bittercress".to_string()] };
    if complicated_condition() {
        p = q;
    }
}
println!("Sproing! What was that?");
```

Selon qu'il `complicated_condition` renvoie `true` ou `false`, soit `p` ou `q` finira par posséder le `Appellation`, avec l'autre non initialisé. L'endroit où il atterrit détermine s'il est déposé avant ou après le `println!`, car il `q` est hors de portée avant le `println!`, et `p` après. Bien qu'une valeur puisse être déplacée d'un endroit à l'autre, Rust ne la supprime qu'une seule fois.

Vous n'aurez généralement pas besoin d'implémenter à `std::ops::Drop` moins que vous ne définissiez un type qui possède des ressources que Rust ne connaît pas déjà. Par exemple, sur les systèmes Unix, la bibliothèque standard de Rust utilise le type suivant en interne pour représenter un descripteur de fichier du système d'exploitation :

```
struct FileDesc {
    fd:c_int,
```

```
}
```

Le `fd` champ de a `FileDesc` est simplement le numéro du descripteur de fichier qui doit être fermé lorsque le programme en a fini avec lui ; `c_int` est un alias pour `i32` . La bibliothèque standard implémente `Drop` pour `FileDesc` comme suit :

```
impl Drop for FileDesc {
    fn drop(&mut self) {
        let _ = unsafe { libc::close(self.fd) };
    }
}
```

Voici le nom Rust de la fonction `libc::close` de la bibliothèque C. `close` Le code Rust peut appeler des fonctions C uniquement dans `unsafe` des blocs, donc la bibliothèque en utilise une ici.

Si un type implémente `Drop` , il ne peut pas implémenter le `Copy` trait. Si un type est `Copy` , cela signifie qu'une simple duplication octet par octet est suffisante pour produire une copie indépendante de la valeur. Mais c'est généralement une erreur d'appeler la même `drop` méthode plus d'une fois sur les mêmes données.

Le prélude standard inclut une fonction pour supprimer une valeur `drop` , mais sa définition est tout sauf magique :

```
fn drop<T>(_x:T) { }
```

En d'autres termes, il reçoit son argument par valeur, prenant possession de l'appelant, puis ne fait rien avec. Rust supprime la valeur `_x` lorsqu'il sort de la portée, comme il le ferait pour toute autre variable.

Taille

Une *tailletype* est celui dont les valeurs ont toutes la même taille en mémoire. Presque tous les types de Rust sont dimensionnés : chacun `u64` prend huit octets, chaque `(f32, f32, f32)` tuple douze. Même les énumérations sont dimensionnées : quelle que soit la variante réellement présente, une énumération occupe toujours suffisamment d'espace pour contenir sa plus grande variante. Et bien que a `Vec<T>` possède un tampon alloué par tas dont la taille peut varier, la `Vec` valeur elle-même est

un pointeur vers le tampon, sa capacité et sa longueur, tout `Vec<T>` comme un type dimensionné.

Tous les types dimensionnés implémentent le `std::marker::Sized` trait, qui n'a pas de méthodes ou de types associés. Rust l'implémente automatiquement pour tous les types auxquels il s'applique ; vous ne pouvez pas l'implémenter vous-même. La seule utilisation de for `Sized` est en tant que borne pour les variables de type : une borne like `T: Sized` nécessite `T` d'être un type dont la taille est connue au moment de la compilation. Les traits de ce type sont appelés *traits marqueurs* , car le langage Rust lui-même les utilise pour marquer certains types comme ayant des caractéristiques intéressantes.

Cependant, Rust a également *quelquestypes* dont les valeurs ne sont pas toutes de la même taille. Par exemple, le type tranche de chaîne `str` (note, sans `&`) n'est pas dimensionné. Les littéraux de chaîne `"diminutive"` et `"big"` sont des références à des `str` tranches qui occupent dix et trois octets. Les deux sont illustrés à la Figure 13-1 . Les types de tranches de tableau comme `[T]` (encore une fois, sans `&`) ne sont pas non plus dimensionnés : une référence partagée comme `&[u8]` peut pointer vers une `[u8]` tranche de n'importe quelle taille. Étant donné que les types `str` et `[T]` désignent des ensembles de valeurs de tailles variables, ce sont des types non dimensionnés.

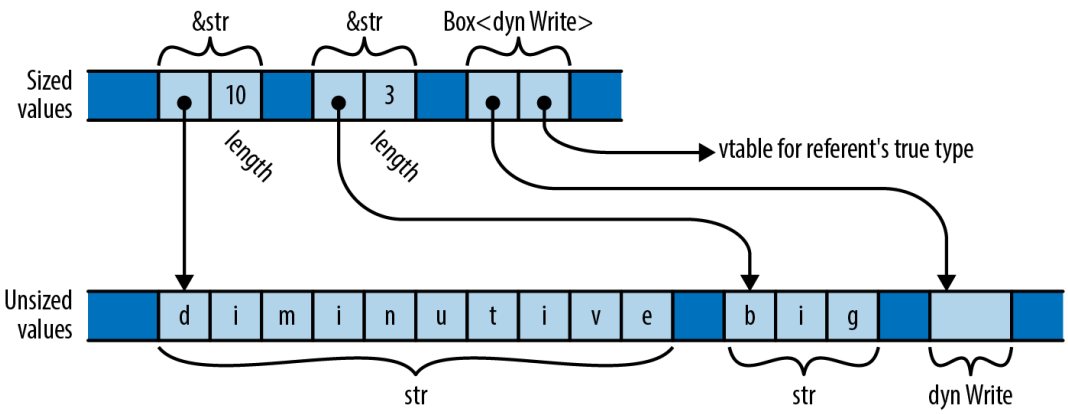

Illustration 13-1. Références à des valeurs non dimensionnées

L'autre type courant de type non dimensionné dans Rust est un `dyn` type, le référent d'un objet trait. Comme nous l'avons expliqué dans "Trait Objects" , un objet trait est un pointeur vers une valeur qui implémente un trait donné. Par exemple, les types `&dyn std::io::Write` et `Box<dyn std::io::Write>` sont des pointeurs vers une valeur qui implémente le `Write` trait. Le référent peut être un fichier ou un socket réseau ou un type de votre choix pour lequel vous avez implémenté `Write` . Puisque l'ensemble des types qui implémentent `Write` est ouvert, `dyn`

`Write` considéré comme un type n'est pas dimensionné : ses valeurs ont des tailles différentes.

Rust ne peut pas stocker de valeurs non dimensionnées dans des variables ni les transmettre en tant qu'arguments. Vous ne pouvez les traiter qu'à l'aide de pointeurs tels que `&str` ou `Box<dyn Write>`, eux-mêmes dimensionnés. Comme le montre la figure 13-1, un pointeur vers une valeur non dimensionnée est toujours un *pointeur gras*, large de deux mots : un pointeur vers une tranche porte également la longueur de la tranche, et un objet trait porte également un pointeur vers une vtable d'implémentations de méthodes.

Les objets de trait et les pointeurs vers les tranches sont bien symétriques. Dans les deux cas, le type manque des informations nécessaires pour l'utiliser : vous ne pouvez pas indexer a `[u8]` sans connaître sa longueur, ni invoquer une méthode sur a `Box<dyn Write>` sans connaître l'implémentation de `Write` appropriée à la valeur spécifique à laquelle il se réfère. Et dans les deux cas, le pointeur gras complète les informations manquantes du type, portant une longueur ou un pointeur vtable. Les informations statiques omises sont remplacées par des informations dynamiques.

Étant donné que les types non dimensionnés sont si limités, la plupart des variables de type génériques doivent être limitées aux `Sized` types. En fait, cela est si souvent nécessaire que c'est le défaut implicite de Rust : si vous écrivez `struct S<T> { ... }`, Rust comprend que vous voulez dire `struct S<T: Sized> { ... }`. Si vous ne souhaitez pas restreindre `T` cette méthode, vous devez explicitement vous désinscrire en écrivant `struct S<T: ?Sized> { ... }`. La `?Sized` syntaxe est spécifique à ce cas et signifie "pas nécessairement `Sized`". Par exemple, si vous écrivez `struct S<T: ?Sized> { b: Box<T> }`, alors Rust vous permettra d'écrire `S<str>` et `S<dyn Write>`, où la boîte devient un gros pointeur, ainsi que `S<i32>` et `S<String>`, où la boîte est un pointeur ordinaire.

Malgré leurs restrictions, les types non dimensionnés rendent le système de type de Rust plus fluide. En lisant la documentation de la bibliothèque standard, vous rencontrerez occasionnellement une `?Sized` borne sur une variable de type ; cela signifie presque toujours que le type donné est uniquement pointé et permet au code associé de fonctionner avec des tranches et des objets de trait ainsi qu'avec des valeurs ordinaires. Lors-

qu'une variable de type a la `?Sized` limite, les gens disent souvent qu'elle est *de taille douteuse* : elle peut être `Sized` , ou non.

Outre les tranches et les objets de trait, il existe un autre type de type non dimensionné. Le dernier champ d'un type de struct (mais seulement son dernier) peut être non dimensionné, et un tel struct est lui-même non dimensionné. Par exemple, un `Rc<T>` pointeur de comptage de références est implémenté en interne en tant que pointeur vers le type privé `RcBox<T>` , qui stocke le nombre de références à côté de `T` . Voici une définition simplifiée de `RcBox` :

```
struct RcBox<T: ?Sized> {
    ref_count: usize,
    value:T,
}
```

Le `value` champ est le `T` vers lequel `Rc<T>` compte les références ; `Rc<T>` déréférence à un pointeur vers ce champ. Le `ref_count` champ contient le nombre de références.

Le real `RcBox` n'est qu'un détail d'implémentation de la bibliothèque standard et n'est pas disponible pour un usage public. Mais supposons que nous travaillions avec la définition précédente. Vous pouvez l'utiliser `RcBox` avec des types dimensionnés, comme `RcBox<String>` ; le résultat est un type de structure dimensionné. Ou vous pouvez l'utiliser avec des types non dimensionnés, comme `RcBox<dyn std::fmt::Display>` (où `Display` est le trait pour les types qui peuvent être formatés par `println!` et des macros similaires) ; `RcBox<dyn Display>` est un type de structure non dimensionné.

Vous ne pouvez pas créer une `RcBox<dyn Display>` valeur directement. Au lieu de cela, vous devez d'abord créer un format ordinaire `RcBox` dont le `value` type implémente `Display` , comme `RcBox<String>` . Rust permet alors de convertir une référence `&RcBox<String>` en référence fat `&RcBox<dyn Display>` :

```
let boxed_lunch: RcBox<String> = RcBox {
    ref_count: 1,
    value:"lunch".to_string()
};

use std:: fmt:: Display;
let boxed_displayable:&RcBox<dyn Display> = &boxed_lunch;
```

Cette conversion se produit implicitement lors du passage de valeurs à des fonctions, vous pouvez donc passer an `&RcBox<String>` à une fonction qui attend un `&RcBox<dyn Display>` :

```
fn display(boxed:&RcBox<dyn Display>) {
    println!("For your enjoyment: {}", &boxed.value);
}

display(&boxed_lunch);
```

Cela produirait la sortie suivante:

```
For your enjoyment: lunch
```

Cloner

Le `std::clone::Clone` traitest pour les types qui peuvent faire des copies d'eux-mêmes. `Clone` est défini comme suit :

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source:&Self) {
        *self = source.clone()
    }
}
```

La `clone` méthode doit construire une copie indépendante de `self` et la renvoyer. Étant donné que le type de retour de cette méthode est `Self` et que les fonctions ne peuvent pas renvoyer de valeurs non dimensionnées, le `Clone` trait lui-même étend le `Sized` trait : cela a pour effet de limiter les `Self` types d'implémentations à `Sized` .

Le clonage d'une valeur implique généralement d'allouer des copies de tout ce qu'elle possède, de sorte qu'une `clone` peut être coûteuse, en temps et en mémoire. Par exemple, le clonage de a `Vec<String>` non seulement copie le vecteur, mais copie également chacun de ses `String` éléments. C'est pourquoi Rust ne se contente pas de cloner automatiquement les valeurs, mais vous oblige à faire un appel de méthode explicite. Les types de pointeurs à comptage de références comme `Rc<T>` et `Arc<T>` sont des exceptions : le clonage de l'un d'entre eux in-

crémente simplement le nombre de références et vous donne un nouveau pointeur.

La `clone_from` méthode `self` se transforme en une copie de `source`. La définition par défaut de `clone_from` simplement cloner `source`, puis la déplace dans `*self`. Cela fonctionne toujours, mais pour certains types, il existe un moyen plus rapide d'obtenir le même effet. Par exemple, supposons que `s` et `t` sont `String`s. L'instruction `s = t.clone();` doit cloner `t`, supprimer l'ancienne valeur de `s`, puis déplacer la valeur clonée dans `s` ; c'est une allocation de tas et une désallocation de tas. Mais si le tampon de tas appartenant à l'original `s` a une capacité suffisante pour contenir `t` le contenu de , aucune allocation ou désallocation n'est nécessaire : vous pouvez simplement copier `t` le texte `s` de dans le tampon de et ajuster la longueur. En code générique, vous devez utiliser `clone_from` dans la mesure du possible pour tirer parti des implémentations optimisées lorsqu'elles sont présentes.

Si votre `Clone` implémentation s'applique simplement `clone` à chaque champ ou élément de votre type, puis construit une nouvelle valeur à partir de ces clones, et que la définition par défaut de `clone_from` est suffisamment bonne, alors Rust l'implémentera pour vous : placez-le simplement `#[derive(Clone)]` au-dessus de votre définition de type.

À peu près tous les types de la bibliothèque standard qui ont du sens pour copier les outils `Clone`. Les types primitifs aiment `bool` et `i32` font. Les types de conteneurs tels que `String`, `Vec<T>` et `HashMap` do également. Certains types n'ont pas de sens à copier, comme `std::sync::Mutex` ; ceux-ci ne sont pas implémentés `Clone`. Certains types comme `std::fs::File` peuvent être copiés, mais la copie peut échouer si le système d'exploitation ne dispose pas des ressources nécessaires ; ces types n'implémentent pas `Clone`, car ils `clone` doivent être infaillibles. Au lieu de cela, `std::fs::File` fournit une `try_clone` méthode qui renvoie un `std::io::Result<File>`, qui peut signaler un échec.

Copie

Au chapitre 4 , nous avons expliqué que, pour la plupart des types, l'affectation déplace les valeurs au lieu de les copier. Le déplacement des valeurs facilite grandement le suivi des ressources qu'ils possèdent. Mais dans "Copy Types : The Exception to Moves" , nous avons souligné l'exception : les types simples qui ne possèdent aucune ressource peuvent être

des `Copy` types, où l'affectation fait une copie de la source, plutôt que de déplacer la valeur et de laisser la source non initialisée .

À ce moment-là, nous avons laissé dans le vague exactement ce qui `Copy` était, mais maintenant nous pouvons vous dire : un type est `Copy` s'il implémente le `std::marker::Copy` trait marqueur, qui est défini comme suit :

```
trait Copy:Clone { }
```

Ceci est certainement facile à mettre en œuvre pour vos propres types :

```
impl Copy for MyType { }
```

Mais comme il `Copy` s'agit d'un trait de marqueur ayant une signification particulière pour le langage, Rust permet à un type de ne s'implémenter `Copy` que si une copie superficielle octet par octet est tout ce dont il a besoin. Les types qui possèdent d'autres ressources, comme les tampons de tas ou les handles du système d'exploitation, ne peuvent pas implémenter `Copy` .

Tout type qui implémente le `Drop` trait ne peut pas être `Copy` . Rust suppose que si un type nécessite un code de nettoyage spécial, il doit également nécessiter un code de copie spécial et ne peut donc pas être `Copy` .

Comme pour `Clone` , vous pouvez demander à Rust de dériver `Copy` pour vous, en utilisant `#[derive(Copy)]` . Vous verrez souvent les deux dérivés à la fois, avec `#[derive(Copy, Clone)]` .

Réfléchissez bien avant de faire un type `Copy` . Bien que cela facilite l'utilisation du type, cela impose de lourdes restrictions à sa mise en œuvre. Les copies implicites peuvent également être coûteuses. Nous expliquons ces facteurs en détail dans "Types de copie : l'exception aux déplacements" .

Deref et DerefMut

Vous pouvez spécifier comment le déréférencementles opérateurs aiment `*` et `.` se comportent sur vos types en implémentant les traits `std::ops::Deref` et . `std::ops::DerefMut` Les types de pointeurs aiment `Box<T>` et `Rc<T>` implémentent ces traits afin qu'ils puissent se

comporter comme le font les types de pointeurs intégrés de Rust. Par exemple, si vous avez une `Box<Complex>` valeur `b`, alors `*b` fait référence à la `Complex` valeur qui `b` pointe vers et `b.re` fait référence à son composant réel. Si le contexte attribue ou emprunte une référence mutable au référent, Rust utilise le `DerefMut` trait (« déréférencer de manière mutable ») ; sinon, l'accès en lecture seule est suffisant et il utilise `Deref`.

Les traits sont définis comme ceci :

```
trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}

trait DerefMut: Deref {
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Les méthodes `deref` et prennent une référence et renvoient une référence. devrait être quelque chose qui contient, possède ou se réfère à : car le type est . Notez que cela s'étend : si vous pouvez déréférencer quelque chose et le modifier, vous devriez certainement pouvoir également lui emprunter une référence partagée. Étant donné que les méthodes renvoient une référence avec la même durée de vie que , reste emprunté aussi longtemps que la référence renvoyée
vit. `deref_mut &Self &Self::Target Target Self Box<Complex> T arget Complex DerefMut Deref &self self`

Les traits `Deref` et `DerefMut` jouent également un autre rôle. Puisque `deref` prend une `&Self` référence et renvoie une `&Self::Target` référence, Rust l'utilise pour convertir automatiquement les références du premier type dans le second. En d'autres termes, si l'insertion d'un `deref` appel permet d'éviter une incompatibilité de type, Rust en insère un pour vous. L' implémentation `DerefMut` permet la conversion correspondante pour les références mutables. Celles-ci sont appelées les *coercitions de deref*: un type est « contraint » à se comporter comme un autre.

Bien que les contraintes de deref ne soient pas quelque chose que vous ne pourriez pas écrire explicitement vous-même, elles sont pratiques :

- Si vous avez une certaine `Rc<String>` valeur `r` et que vous souhaitez l'appliquer `String::find`, vous pouvez simplement écrire

`r.find('?')`, au lieu de `(*r).find('?')` : l'appel de méthode emprunte implicitement `r` et `&Rc<String>` contraint à `&String`, car `Rc<T>` implémente `Deref<Target=T>`.

- Vous pouvez utiliser des méthodes comme `split_at` on `String` values, même s'il `split_at` s'agit d'une méthode de `str` type slice, car `String` implements `Deref<Target=str>`. Il n'est pas nécessaire `String` de réimplémenter toutes `str` les méthodes de , puisque vous pouvez contraindre a à `&str` partir de `&String`.
- Si vous avez un vecteur d'octets `v` et que vous voulez le passer à une fonction qui attend une tranche d'octet `&[u8]`, vous pouvez simplement le passer `&v` comme argument, puisque `Vec<T>` implements `Deref<Target=[T]>`.

Rust appliquera successivement plusieurs coercitions de deref si nécessaire. Par exemple, en utilisant les coercions mentionnées précédemment, vous pouvez appliquer `split_at` directement à un `Rc<String>`, puisque `&Rc<String>` déréférence à `&String`, qui déréférence à `&str`, qui a la `split_at` méthode.

Par exemple, supposons que vous ayez le type suivant :

```
struct Selector<T> {
    /// Elements available in this `Selector`.
    elements:Vec<T>,

    /// The index of the "current" element in `elements`. A `Selector`
    /// behaves like a pointer to the current element.
    current:usize
}
```

Pour faire en sorte que le `Selector` comportement soit conforme au commentaire de la documentation, vous devez implémenter `Deref` et `DerefMut` pour le type :

```
use std:: ops::{Deref, DerefMut};

impl<T> Deref for Selector<T> {
    type Target = T;
    fn deref(&self) ->&T {
        &self.elements[self.current]
    }
}

impl<T> DerefMut for Selector<T> {
```

```
        fn deref_mut(&mut self) ->&mut T {
            &mut self.elements[self.current]
        }
    }
```

Compte tenu de ces implémentations, vous pouvez utiliser un `Selector` comme ceci:

```
    let mut s = Selector { elements: vec!['x', 'y', 'z'],
                           current:2 };

    // Because `Selector` implements `Deref`, we can use the `*` operator to
    // refer to its current element.
    assert_eq!(*s, 'z');

    // Assert that 'z' is alphabetic, using a method of `char` directly on a
    // `Selector`, via deref coercion.
    assert!(s.is_alphabetic());

    // Change the 'z' to a 'w', by assigning to the `Selector`'s referent.
    *s = 'w';

    assert_eq!(s.elements, ['x', 'y', 'w']);
```

Les traits `Deref` et `DerefMut` sont conçus pour implémenter des types de pointeurs intelligents, tels que `Box` , `Rc` , et `Arc` , et des types qui servent de versions propriétaires de quelque chose que vous utiliseriez aussi fréquemment par référence, la manière `Vec<T>` et `String` servant de versions propriétaires de `[T]` et `str` . Vous ne devez pas implémenter `Deref` et `DerefMut` pour un type juste pour faire `Target` apparaître automatiquement les méthodes du type, de la même manière que les méthodes d'une classe de base C++ sont visibles sur une sous-classe. Cela ne fonctionnera pas toujours comme prévu et peut être source de confusion lorsque cela tourne mal.

Les coercitions deref s'accompagnent d'une mise en garde qui peut prêter à confusion : Rust les applique pour résoudre les conflits de type, mais pas pour satisfaire les limites sur les variables de type. Par exemple, le code suivant fonctionne correctement :

```
    let s = Selector { elements: vec!["good", "bad", "ugly"],
                       current:2 };
```

```
fn show_it(thing:&str) { println!("{}", thing); }
show_it(&s);
```

Dans l'appel `show_it(&s)`, Rust voit un argument de type `&Selector<&str>` et un paramètre de type `&str`, trouve l'`Deref<Target=str>` implémentation et réécrit l'appel en tant que `show_it(s.deref())`, juste au besoin.

Cependant, si vous changez `show_it` en une fonction générique, Rust n'est soudainement plus coopératif :

```
use std:: fmt:: Display;
fn show_it_generic<T: Display>(thing:T) { println!("{}", thing); }
show_it_generic(&s);
```

La rouille se plaint :

```
error: `Selector<&str>` doesn't implement `std::fmt::Display`
   |
31 |  show_it_generic(&s);
   |                  ^^
   |                  |
   |                  `Selector<&str>` cannot be formatted with
   |                  the default formatter
   |                  help: consider adding dereference here: `&*s`
   |
note: required by a bound in `show_it_generic`
   |
30 |  fn show_it_generic<T: Display>(thing: T) { println!("{}", thing);
   |                        ^^^^^^^ required by this bound
   |                                in `show_it_generic`
```

Cela peut être déconcertant : comment le fait de rendre une fonction générique pourrait-il introduire une erreur ? Certes, `Selector<&str>` ne s'implémente pas `Display`, mais il déréférence à `&str`, ce qui est certainement le cas.

Puisque vous passez un argument de type `&Selector<&str>` et que le type de paramètre de la fonction est `&T`, la variable de type `T` doit être `Selector<&str>`. Ensuite, Rust vérifie si la borne `T: Display` est satisfaite : puisqu'il n'applique pas de coercitions de deref pour satisfaire les bornes sur les variables de type, cette vérification échoue.

Pour contourner ce problème, vous pouvez épeler la coercition à l'aide de l' `as` opérateur:

```
show_it_generic(&s as &str);
```

Ou, comme le suggère le compilateur, vous pouvez forcer la coercition avec `&*` :

```
show_it_generic(&*s);
```

Défaut

Certains types ont un défaut raisonnablement évidentvaleur : le vecteur ou la chaîne par défaut est vide, le nombre par défaut est zéro, la valeur par défaut `Option` est `None` , etc. Des types comme celui-ci peuvent implémenter le `std::default::Default` trait :

```
trait Default {
    fn default() ->Self;
}
```

La `default` méthode renvoie simplement une nouvelle valeur de type `Self` . `String` L'implémentation de `Default` est simple :

```
impl Default for String {
    fn default() -> String {
        String::new()
    }
}
```

Tous les types de collection de Rust — `Vec` , `HashMap` , `BinaryHeap` , etc. — implémentent `Default` , avec des `default` méthodes qui renvoient une collection vide. Ceci est utile lorsque vous devez créer une collection de valeurs, mais que vous souhaitez laisser votre appelant décider exactement du type de collection à créer. Par exemple, la méthode `Iterator` du trait `partition` divise les valeurs produites par l'itérateur en deux collections, en utilisant une fermeture pour décider où va chaque valeur :

```
use std:: collections:: HashSet;
let squares = [4, 9, 16, 25, 36, 49, 64];
let (powers_of_two, impure):(HashSet<i32>, HashSet<i32>)
    = squares.iter().partition(|&n| n & (n-1) == 0);

assert_eq!(powers_of_two.len(), 3);
assert_eq!(impure.len(), 4);
```

La fermeture `|&n| n & (n-1) == 0` utilise un peu de bricolage pour reconnaître les nombres qui sont des puissances de deux, et `partition` l'utilise pour produire deux `HashSet`s. Mais bien sûr, `partition` n'est pas spécifique à `HashSet`s; vous pouvez l'utiliser pour produire n'importe quel type de collection, tant que le type de collection implémente `Default`, pour produire une collection vide pour commencer, et `Extend<T>`, pour ajouter un `T` à la collection. `String` implémente `Default` et `Extend<char>`, vous pouvez donc écrire :

```
let (upper, lower):(String, String)
    = "Great Teacher Onizuka".chars().partition(|&c| c.is_uppercase());
assert_eq!(upper, "GTO");
assert_eq!(lower, "reat eacher nizuka");
```

Une autre utilisation courante de `Default` consiste à produire des valeurs par défaut pour les structures qui représentent une grande collection de paramètres, dont la plupart n'auront généralement pas besoin d'être modifiés. Par exemple, la `glium` caisse fournit des liaisons Rust pour la puissante et complexe bibliothèque graphique OpenGL. La `glium::DrawParameters` structure comprend 24 champs, chacun contrôlant un détail différent de la façon dont OpenGL doit rendre certains éléments graphiques. La `glium draw` fonction attend une `DrawParameters` structure comme argument. Depuis `DrawParameters` implements `Default`, vous pouvez en créer un à transmettre à `draw`, en mentionnant uniquement les champs que vous souhaitez modifier :

```
let params = glium:: DrawParameters {
    line_width: Some(0.02),
    point_size: Some(0.02),
    .. Default::default()
};

target.draw(..., &params).unwrap();
```

Cela appelle `Default::default()` à créer une `DrawParameters` valeur initialisée avec les valeurs par défaut pour tous ses champs, puis utilise la `..` syntaxe des structures pour en créer une nouvelle avec les champs `line_width` et `point_size` modifiés, prêt à être transmis à `target.draw`.

Si un type `T` implémente `Default`, alors la bibliothèque standard implémente `Default` automatiquement pour `Rc<T>`, `Arc<T>`, `Box<T>`, `Cell<T>`, `RefCell<T>`, `Cow<T>`, `Mutex<T>`, et `RwLock<T>`. La valeur par défaut du type `Rc<T>`, par exemple, est un `Rc` pointant vers la valeur par défaut du type `T`.

Si tous les types d'éléments d'un type de tuple implémentent `Default`, alors le type de tuple le fait aussi, par défaut un tuple contenant la valeur par défaut de chaque élément.

Rust n'implémente pas implicitement `Default` pour les types de structure, mais si tous les champs d'une structure implémentent `Default`, vous pouvez implémenter `Default` pour la structureautomatiquement en utilisant `#[derive(Default)]`.

AsRef et AsMut

Lorsqu'un genremet en œuvre `AsRef<T>`, cela signifie que vous pouvez lui emprunter un `&T` efficacement. `AsMut` est l'analoguepour les références mutables. Leurs définitions sont les suivantes :

```
trait AsRef<T: ?Sized> {
    fn as_ref(&self) ->&T;
}

trait AsMut<T: ?Sized> {
    fn as_mut(&mut self) ->&mut T;
}
```

Ainsi, par exemple, `Vec<T>` implements `AsRef<[T]>`, et `String` implements `AsRef<str>`. Vous pouvez également emprunter `String` le contenu d'un sous forme de tableau d'octets, donc `String` implémente `AsRef<[u8]>` également.

`AsRef` est généralement utilisé pour rendre les fonctions plus flexibles dans les types d'arguments qu'elles acceptent. Par exemple, la

`std::fs::File::open` fonction est déclarée comme ceci :

```
fn open<P: AsRef<Path>>(path: P) ->Result<File>
```

Ce qu'il faut `open` vraiment, c'est un `&Path`, le type représentant un chemin de système de fichiers. Mais avec cette signature, `open` accepte tout ce à quoi il peut emprunter `&Path`, c'est-à-dire tout ce qui implémente `AsRef<Path>`. Ces types incluent `String` et `str`, les types de chaîne d'interface du système d'exploitation `OsString` et `OsStr`, et bien sûr `PathBuf` et `Path` ; voir la documentation de la bibliothèque pour la liste complète. C'est ce qui vous permet de passer des littéraux de chaîne à `open` :

```
let dot_emacs = std:: fs:: File::open("/home/jimb/.emacs")?;
```

Toutes les fonctions d'accès au système de fichiers de la bibliothèque standard acceptent les arguments de chemin de cette façon. Pour les appelants, l'effet ressemble à celui d'une fonction surchargée en C++, bien que Rust adopte une approche différente pour déterminer quels types d'arguments sont acceptables.

Mais cela ne peut pas être toute l'histoire. Un littéral de chaîne est un `&str`, mais le type qui implémente `AsRef<Path>` est `str`, sans un `&`. Et comme nous l'avons expliqué dans "Deref et DerefMut", Rust n'essaie pas de coercitions de deref pour satisfaire les limites de variables de type, donc elles ne seront pas utiles ici non plus.

Heureusement, la bibliothèque standard inclut l'implémentation de couverture :

```
impl<'a, T, U> AsRef<U> for &'a T
    where T: AsRef<U>,
          T: ?Sized, U: ?Sized
{
    fn as_ref(&self) ->&U {
        (*self).as_ref()
    }
}
```

En d'autres termes, pour tous les types `T` et `U`, si `T: AsRef<U>`, alors `&T: AsRef<U>` également : suivez simplement la référence et procédez comme avant. En particulier, depuis `str: AsRef<Path>`, alors `&str:`

`AsRef<Path>` aussi. Dans un sens, c'est un moyen d'obtenir une forme limitée de coercition de deref lors de la vérification des `AsRef` bornes sur les variables de type.

Vous pouvez supposer que si un type implémente `AsRef<T>` , il doit également implémenter `AsMut<T>` . Cependant, il y a des cas où cela n'est pas approprié. Par exemple, nous avons mentionné que `String` implements `AsRef<[u8]>` ; cela a du sens, car chacun `String` a certainement un tampon d'octets qui peut être utile pour accéder en tant que données binaires. Cependant, `String` garantit en outre que ces octets sont un codage UTF-8 bien formé du texte Unicode ; s'il était `String` implémenté `AsMut<[u8]>` , cela permettrait aux appelants de changer les `String` octets de pour tout ce qu'ils voulaient, et vous ne pourriez plus faire confiance à a `String` pour être UTF-8 bien formé. L' implémentation d'un type n'a de sens que `AsMut<T>` si la modification du donné `T` ne peut pas violer les invariants du type.

Bien que `AsRef` et `AsMut` soient assez simples, fournir des traits génériques standard pour la conversion de référence évite la prolifération de traits de conversion plus spécifiques. Vous devriez éviter de définir vos propres `AsFoo` traits alors que vous pourriez simplement implémenter `AsRef<Foo>` .

Emprunter et emprunterMut

Le `std::borrow::Borrow` traitest similaire à `AsRef` : si un type implémente `Borrow<T>` , alors sa `borrow` méthode lui emprunte efficacement `&T` a. Mais `Borrow` impose plus de restrictions : un type ne doit être implémenté `Borrow<T>` que lorsqu'un `&T` hachage et se compare de la même manière que la valeur à laquelle il est emprunté. (Rust ne l'applique pas ; c'est juste l'intention documentée du trait.) Cela rend `Borrow` utile le traitement des clés dans les tables de hachage et les arbres ou lors du traitement des valeurs qui seront hachées ou comparées pour une autre raison.

Cette distinction est importante lorsque vous empruntez à `String` s, par exemple : `String` implements `AsRef<str>` , `AsRef<[u8]>` et `AsRef<Path>` , mais ces trois types de cibles auront généralement des valeurs de hachage différentes. Seule la `&str` tranche est garantie de hacher comme l'équivalent `String` , donc `String` implémente uniquement `Borrow<str>` .

`Borrow` la définition de est identique à celle de `AsRef` ; seuls les noms ont été modifiés :

```
trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) ->&Borrowed;
}
```

`Borrow` est conçu pour traiter une situation spécifique avec des tables de hachage génériques et d'autres types de collections associatives. Par exemple, supposons que vous ayez un `std::collections ::HashMap<String, i32>` mappage de chaînes à des nombres. Les clés de cette table sont `String` s ; chaque entrée en possède un. Quelle devrait être la signature de la méthode qui recherche une entrée dans cette table ? Voici une première tentative :

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: K) ->Option<&V> { ... }
}
```

Cela a du sens : pour rechercher une entrée, vous devez fournir une clé du type approprié pour la table. Mais dans ce cas, `K` est `String` ; cette signature vous obligerait à passer a `String` par valeur à chaque appel à `get` , ce qui est clairement un gaspillage. Vous avez vraiment juste besoin d'une référence à la clé:

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: &K) ->Option<&V> { ... }
}
```

C'est un peu mieux, mais maintenant vous devez passer la clé en tant que `&String` , donc si vous vouliez rechercher une chaîne constante, vous devriez écrire :

```
hashtable.get(&"twenty-two".to_string())
```

C'est ridicule : il alloue un `String` tampon sur le tas et y copie le texte, juste pour pouvoir l'emprunter en tant que `&String` , le passer à `get` , puis le déposer.

Il devrait être suffisant pour transmettre tout ce qui peut être haché et comparé avec notre type de clé ; a `&str` devrait être parfaitement adéquat, par exemple. Voici donc la dernière itération, qui correspond à ce que vous trouverez dans la bibliothèque standard :

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
        where K: Borrow<Q>,
              Q:Eq + Hash
    { ... }
}
```

En d'autres termes, si vous pouvez emprunter la clé d'une entrée en tant que `&Q` et que la référence résultante hache et compare exactement comme le ferait la clé elle-même, alors il `&Q` devrait clairement s'agir d'un type de clé acceptable. Depuis `String` implémente `Borrow<str>` et `Borrow<String>` , cette version finale de `get` vous permet de passer soit `&String` ou `&str` comme clé, selon vos besoins.

`Vec<T>` et mettre en `[T: N]` œuvre `Borrow<[T]>` . Chaque type de type chaîne permet d'emprunter son type de tranche correspondant : `String` implements `Borrow<str>` , `PathBuf` implements `Borrow<Path>` , etc. Et tous les types de collection associatifs de la bibliothèque standard utilisent `Borrow` pour décider quels types peuvent être passés à leurs fonctions de recherche.

La bibliothèque standard inclut une implémentation globale afin que chaque type `T` puisse être emprunté à lui-même : `T: Borrow<T>` . Cela garantit qu'il `&K` s'agit toujours d'un type acceptable pour rechercher des entrées dans un fichier `HashMap<K, V>` .

Par commodité, chaque `&mut T` type implémente également `Borrow<T>` , renvoyant une référence partagée `&T` comme d'habitude. Cela vous permet de passer des références mutables aux fonctions de recherche de collection sans avoir à réemprunter une référence partagée, émulant la coercition implicite habituelle de Rust des références mutables aux références partagées.

Le `BorrowMut` trait est l'analogue de `Borrow` pour les références mutables :

```
trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) ->&mut Borrowed;
}
```

Les mêmes attentes décrites pour `Borrow` s'appliquent `BorrowMut` également à.

De et vers

Les `std::convert::From` et `std::convert::Into` traitsreprésentent les conversions qui consomment une valeur d'un type et renvoient une valeur d'un autre. Alors que les traits `AsRef` et empruntent une référence d'un type à un autre et s'approprient leur argument, le transforment, puis rendent la propriété du résultat à l'appelant. `AsMut From Into`

Leurs définitions sont bien symétriques :

```
trait Into<T>: Sized {
    fn into(self) ->T;
}

trait From<T>: Sized {
    fn from(other: T) ->Self;
}
```

La bibliothèque standard implémente automatiquement la conversion triviale de chaque type vers lui-même : chaque type `T` implémente `From<T>` and `Into<T>` .

Bien que les traits fournissent simplement deux façons de faire la même chose, ils se prêtent à des utilisations différentes.

Vous utilisez généralement `Into` pour rendre vos fonctions plus flexibles dans les arguments qu'elles acceptent. Par exemple, si vous écrivez :

```
use std:: net:: Ipv4Addr;
fn ping<A>(address: A) -> std:: io:: Result<bool>
    where A:Into<Ipv4Addr>
{
    let ipv4_address = address.into();
    ...
}
```

then `ping` peut accepter non seulement `Ipv4Addr` a comme argument, mais aussi a `u32` ou un `[u8; 4]` tableau, puisque ces types implémentent commodément `Into<Ipv4Addr>` . (Il est parfois utile de traiter une adresse IPv4 comme une seule valeur 32 bits ou un tableau de 4 octets.) Comme la seule chose `ping` connue `address` est qu'elle implémente `Into<Ipv4Addr>` , il n'est pas nécessaire de spécifier le type que vous voulez lorsque vous appelez `into` ; il n'y en a qu'un qui pourrait fonctionner, alors l'inférence de type le remplit pour vous.

Comme `AsRef` dans la section précédente, l'effet ressemble beaucoup à celui de la surcharge d'une fonction en C++. Avec la définition de `ping` from before, nous pouvons faire n'importe lequel de ces appels :

```
println!("{:?}", ping(Ipv4Addr::new(23, 21, 68, 141))); // pass an Ipv4Ad
println!("{:?}", ping([66, 146, 219, 98]));             // pass a [u8; 4]
println!("{:?}", ping(0xd076eb94_u32));                 // pass a u32
```

Le `From` trait, cependant, joue un rôle différent. La `from` méthode sert de constructeur générique pour produire une instance d'un type à partir d'une autre valeur unique. Par exemple, plutôt que d' `Ipv4Addr` avoir deux méthodes nommées `from_array` et `from_u32` , il implémente simplement `From<[u8;4]>` et `From<u32>` , ce qui nous permet d'écrire :

```
let addr1 = Ipv4Addr:: from([66, 146, 219, 98]);
let addr2 = Ipv4Addr::from(0xd076eb94_u32);
```

Nous pouvons laisser l'inférence de type déterminer quelle implémentation s'applique.

Etant donné une `From` implémentation appropriée, la bibliothèque standard implémente automatiquement le `Into` trait correspondant. Lorsque vous définissez votre propre type, s'il a des constructeurs à argument unique, vous devez les écrire en tant qu'implémentations de `From<T>` pour les types appropriés ; vous obtiendrez `Into` gratuitement les implémentations correspondantes.

Étant donné que les méthodes de conversion `from` et `into` s'approprient leurs arguments, une conversion peut réutiliser les ressources de la valeur d'origine pour construire la valeur convertie. Par exemple, supposons que vous écriviez :

```
let text = "Beautiful Soup".to_string();
let bytes:Vec<u8> = text.into();
```

L'implémentation de `Into<Vec<u8>>` for `String` prend simplement le `String` tampon de tas de et le réutilise, tel quel, comme tampon d'élément du vecteur renvoyé. La conversion n'a pas besoin d'allouer ou de copier le texte. C'est un autre cas où les déplacements permettent des implémentations efficaces.

Ces conversions fournissent également un bon moyen d'assouplir une valeur d'un type contraint en quelque chose de plus flexible, sans affaiblir les garanties du type contraint. Par exemple, a `String` garantit que son contenu est toujours valide en UTF-8 ; ses méthodes de mutation sont soigneusement restreintes pour s'assurer que rien de ce que vous pouvez faire n'introduira jamais un mauvais UTF-8. Mais cet exemple "rétrograde" efficacement a `String` en un bloc d'octets simples avec lequel vous pouvez faire tout ce que vous voulez : vous allez peut-être le compresser ou le combiner avec d'autres données binaires qui ne sont pas UTF-8. Parce que `into` prend son argument par valeur, `text` n'est plus initialisé après la conversion, ce qui signifie que nous pouvons accéder librement au `String` tampon du premier sans pouvoir corrompre aucun fichier existant `String` .

Cependant, les conversions bon marché ne font pas partie du contrat de `Into` and `From` . Alors `AsRef` que les conversions et `AsMut` sont censées être bon marché, `From` et `Into` les conversions peuvent allouer, copier ou autrement traiter le contenu de la valeur. Par exemple, `String` implements `From<&str>` , qui copie la tranche de chaîne dans un nouveau tampon alloué par tas pour le `String` . Et `std::collections::BinaryHeap<T>` implémente `From<Vec<T>>` , qui compare et réordonne les éléments en fonction des exigences de son algorithme.

L' `?` opérateur utilise `From` et `Into` pour aider à nettoyer le code dans les fonctions qui pourraient échouer de plusieurs façons en convertissant automatiquement des types d'erreurs spécifiques en types généraux si nécessaire.

Par exemple, imaginez un système qui doit lire des données binaires et en convertir une partie à partir de nombres en base 10 écrits sous forme de texte UTF-8. Cela signifie utiliser `std::str::from_utf8` et l' `FromStr` implémentation de `i32` , qui peuvent chacune renvoyer des er-

reurs de types différents. En supposant que nous utilisions les types `GenericError` et que nous avons définis au chapitre 7 lors de la discussion sur la gestion des erreurs, l' opérateur effectuera la conversion pour nous : `GenericResult ?`

```
type GenericError = Box<dyn std:: error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;

fn parse_i32_bytes(b: &[u8]) -> GenericResult<i32> {
        Ok(std:: str:: from_utf8(b)?.parse::<i32>()?)
}
```

Comme la plupart des types d'erreurs, `Utf8Error` et `ParseIntError` implémentez le `Error` trait, et la bibliothèque standard nous donne une couverture `From impl` pour convertir tout ce qui implémente `Error` en un `Box<dyn Error>` , qui `?` utilise automatiquement :

```
impl<'a, E: Error + Send + Sync + 'a> From<E>
  for Box<dyn Error + Send + Sync + 'a> {
    fn from(err: E) -> Box<dyn Error + Send + Sync + 'a> {
        Box::new(err)
    }
}
```

Cela transforme ce qui aurait été une fonction assez volumineuse avec deux `match` instructions en une seule ligne.

Avant `From` et `Into` ont été ajoutés à la bibliothèque standard, le code Rust était plein de traits de conversion et de méthodes de construction ad hoc, chacun spécifique à un seul type. `From` et `Into` codifiez les conventions que vous pouvez suivre pour rendre vos types plus faciles à utiliser, puisque vos utilisateurs les connaissent déjà. D'autres bibliothèques et le langage lui-même peuvent également s'appuyer sur ces caractéristiques comme moyen canonique et standardisé d'encoder les conversions.

`From` et `Into` sont des traits infaillibles - leur API exige que les conversions n'échouent pas. Malheureusement, de nombreuses conversions sont plus complexes que cela. Par exemple, de grands nombres entiers comme `i64` peuvent stocker des nombres bien plus grands que `i32` , et convertir un nombre comme `2_000_000_000_000i64` en `i32` n'a pas beaucoup de sens sans quelques informations supplémentaires. Faire une simple conversion au niveau du bit, dans laquelle les 32 premiers bits sont supprimés, ne donne pas souvent le résultat que nous espérions :

```
let huge = 2_000_000_000_000i64;
let smaller = huge as i32;
println!("{}", smaller); // -1454759936
```

Il existe de nombreuses options pour gérer cette situation. Selon le contexte, une telle conversion « enveloppante » peut être appropriée. D'autre part, des applications telles que le traitement numérique du signal et les systèmes de contrôle peuvent souvent se contenter d'une conversion "saturante", dans laquelle les nombres supérieurs à la valeur maximale possible sont limités à cette valeur maximale..

TryFrom et TryInto

Puisqu'il n'est pas clair comment une telle conversiondevrait se comporter, Rust n'implémente pas `From<i64>` for `i32`, ou toute autre conversion entre types numériques qui perdrait des informations. Au lieu de cela, `i32` implémente `TryFrom<i64>`. `TryFrom` et `TryInto` sont les cousins faillibles de `From` et `Into` et sont pareillement réciproques ; moyens de mise en œuvre `TryFrom` qui `TryInto` sont également mis en œuvre.

Leurs définitions sont seulement un peu plus complexes que `From` et `Into`.

```
pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

pub trait TryInto<T>: Sized {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

La `try_into()` méthode nous donne un `Result`, afin que nous puissions choisir ce qu'il faut faire dans le cas exceptionnel, comme un nombre trop grand pour tenir dans le type résultant :

```
// Saturate on overflow, rather than wrapping
let smaller: i32 = huge.try_into().unwrap_or(i32::MAX);
```

Si nous voulons également traiter le cas négatif, nous pouvons utiliser la `unwrap_or_else()` méthode de `Result` :

```
let smaller: i32 = huge.try_into().unwrap_or_else(|_|{
    if huge >= 0 {
        i32:: MAX
    } else {
        i32::MIN
    }
});
```

La mise en œuvre de conversions faillibles pour vos propres types est également facile. Le `Error` type peut être aussi simple ou aussi complexe qu'une application particulière l'exige. La bibliothèque standard utilise une structure vide, ne fournissant aucune information autre que le fait qu'une erreur s'est produite, puisque la seule erreur possible est un débordement. D'un autre côté, les conversions entre des types plus complexes peuvent vouloir renvoyer plus d'informations :

```
impl TryInto<LinearShift> for Transform {
    type Error = TransformError;

    fn try_into(self) -> Result<LinearShift, Self:: Error> {
        if !self.normalized() {
            return Err(TransformError::NotNormalized);
        }
        ...
    }
}
```

Où `From` et `Into` relient les types avec des conversions simples, `TryFrom` et `TryInto` étendent la simplicité de `From` et `Into` les conversions avec la gestion expressive des erreurs offerte par `Result` . Ces quatre traits peuvent être utilisés ensemble pour relier plusieurs types dans une seule caisse.

ÀOwned

Compte tenu d'une référence, la manière habituelle de produire une propriétécopie de son référent est d'appeler `clone` , en supposant que le type implémente `std::clone::Clone` . Mais que se passe-t-il si vous voulez cloner a `&str` ou a `&[i32]` ? Ce que vous voulez probablement,

c'est a `String` ou a `Vec<i32>`, mais `Clone` la définition de ne le permet pas : par définition, le clonage de a `&T` doit toujours renvoyer une valeur de type `T`, et `str` et `[u8]` ne sont pas dimensionnés ; ce ne sont même pas des types qu'une fonction pourrait renvoyer.

Le `std::borrow::ToOwned` trait fournit un moyen légèrement plus lâche de convertir une référence en une valeur possédée :

```
trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;
}
```

Contrairement à `clone`, qui doit retourner exactement `Self`, `to_owned` peut retourner tout ce à quoi vous pourriez emprunter `&Self` : le `Owned` type doit implémenter `Borrow<Self>`. Vous pouvez emprunter a `&[T]` à a `Vec<T>`, donc `[T]` vous pouvez mettre en œuvre `ToOwned<Owned=Vec<T>>`, tant que met en `T` œuvre `Clone`, afin que nous puissions copier les éléments de la tranche dans le vecteur. De même, `str` implements `ToOwned<Owned=String>`, `Path` implements `ToOwned<Owned=PathBuf>`, etc.

Emprunter et posséder au travail : la vache humble

Faire bon usage de Rust implique de réfléchir à des questions de propriété, comme si une fonction doit recevoir un paramètre par référence ou par valeur. Habituellement, vous pouvez choisir une approche ou l'autre, et le type de paramètre reflète votre décision. Mais dans certains cas, vous ne pouvez pas décider d'emprunter ou de devenir propriétaire tant que le programme n'est pas en cours d'exécution ; le `std::borrow::Cow` genre(pour "cloner en écriture") fournit un moyen de le faire.

Sa définition est présentée ici :

```
enum Cow<'a, B: ?Sized>
    where B: ToOwned
{
    Borrowed(&'a B),
```

```
        Owned(<B as ToOwned>::Owned),
    }
```

A `Cow<B>` soit emprunte une référence partagéeà a `B` ou possède une valeur à laquelle on pourrait emprunter une telle référence. Depuis `Cow` implements `Deref`, vous pouvez appeler des méthodes dessus comme s'il s'agissait d'une référence partagée à a `B` : si c'est `Owned`, il emprunte une référence partagée à la valeur possédée ; et si c'est `Borrowed`, il distribue simplement la référence qu'il détient.

Vous pouvez également obtenir une référence mutable à `Cow` la valeur de a en appelant sa `to_mut` méthode, qui renvoie a `&mut B`. Si le `Cow` se trouve être `Cow::Borrowed`, `to_mut` appelle simplement la `to_owned` méthode de la référence pour obtenir sa propre copie du référent, change le `Cow` en a `Cow::Owned`, et emprunte une référence mutable à la valeur nouvellement possédée. Il s'agit du comportement de « clonage en écriture » auquel le nom du type fait référence.

De même, `Cow` a une `into_owned` méthode qui promeut la référence à une valeur possédée, si nécessaire, puis la renvoie, transférant la propriété à l'appelant et consommant le `Cow` dans le processus.

Une utilisation courante de `Cow` consiste à renvoyer soit une constante de chaîne allouée statiquement, soit une chaîne calculée. Par exemple, supposons que vous deviez convertir une énumération d'erreur en message. La plupart des variantes peuvent être gérées avec des chaînes fixes, mais certaines d'entre elles ont des données supplémentaires qui doivent être incluses dans le message. Vous pouvez retourner un `Cow<'static, str>` :

```
use std:: path:: PathBuf;
use std:: borrow:: Cow;
fn describe(error: &Error) -> Cow<'static, str> {
    match *error {
        Error:: OutOfMemory => "out of memory".into(),
        Error:: StackOverflow => "stack overflow".into(),
        Error:: MachineOnFire => "machine on fire".into(),
        Error:: Unfathomable => "machine bewildered".into(),
        Error::FileNotFound(ref path) => {
            format!("file not found: {}", path.display()).into()
        }
    }
}
```

Ce code utilise `Cow` l'implémentation de `Into` pour construire les valeurs. La plupart des bras de cette `match` instruction renvoient une `Cow::Borrowed` référence à une chaîne allouée statiquement. Mais lorsque nous obtenons une `FileNotFound` variante, nous l'utilisons `format!` pour construire un message incorporant le nom de fichier donné. Ce bras de l' `match` instruction produit une `Cow::Owned` valeur.

Les appelants de `describe` qui n'ont pas besoin de changer la valeur peuvent simplement traiter le `Cow` comme un `&str` :

```
println!("Disaster has struck: {}", describe(&error));
```

Les appelants qui ont besoin d'une valeur possédée peuvent facilement en produire une :

```
let mut log: Vec<String> = Vec::new();
...
log.push(describe(&error).into_owned());
```

L'utilisation des `Cow` aides `describe` et de ses appelants reporte l'attribution jusqu'au moment où cela devient nécessaire.