Chapitre 13. Caractéristiques d'utilité

> *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 chapitre décrit ce que nous appelons les traits « utilitaires » de Rust, un sac à main de divers traits de la bibliothèque standard qui ont suffisamment d'impact sur la façon dont Rust est écrit pour que vous deviez les connaître afin d'écrire du code idiomatique et de concevoir des interfaces publiques pour vos caisses que les utilisateurs jugeront correctement « rustiques ». Ils se répartissent en trois grandes catégories :

Traits d'extension de la langue

Tout comme les traits de surcharge d'opérateur que nous avons couverts 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 vos propres types plus étroitement avec le langage. Ceux-ci incluent , et , et les traits de conversion et . Nous les décrirons dans ce chapitre. `Drop` `Deref` `DerefMut` `From` `Into`

Traits de marqueur

Ce sont des traits principalement utilisés pour lier des variables de type génériques afin d'exprimer des contraintes que vous ne pouvez pas capturer autrement. Ceux-ci incluent et . `Sized` `Copy`

Traits de vocabulaire public

Ceux-ci n'ont pas d'intégration de compilateur magique; vous pouvez définir des traits équivalents dans votre propre code. Mais ils servent l'objectif important de définir des solutions conventionnelles pour des problèmes communs. Ceux-ci sont particulièrement précieux dans les interfaces publiques entre les caisses et les modules: en réduisant les variations inutiles, ils rendent les interfaces plus faciles à comprendre, mais ils augmentent également la probabilité que les fonctionnalités de différentes caisses puissent simplement être branchées ensemble directement, sans code de colle standard ou personnalisé. Ceux-ci comprennent , les traits d'emprunt de référence , , et ; les caractères de conversion faillibles et ; et le trait, une généralisation de

```
.Default AsRef AsMut Borrow BorrowMut TryFrom TryInto T
oOwned Clone
```

Celles-ci sont résumées dans le tableau 13-1.

Tableau 13-1. Résumé des traits d'utilité

Trait	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 effectuant une copie octet par octet de la mémoire contenant la valeur.
`Deref` et `DerefMut`	Caractéristiques des types de pointeurs intelligents.
`Default`	Types qui ont une « valeur par défaut » raisonnable.
`AsRef` et `AsMut`	Caractéristiques de conversion pour emprunter un type de référence à un autre.
`Emprunter` et `EmprunterMut`	Traits de conversion, tels que /, mais garantissant en outre un hachage, un ordre et une égalité cohérents. `AsRef AsMut`
`De` et `vers`	Caractéristiques de conversion pour transformer un type de valeur en un autre.

Trait	Description
`TryFrom` et `TryInto`	Caractéristiques de conversion pour transformer un type de valeur en un autre, pour les transformations qui pourraient échouer.
`ToOwned`	Caractéristique de conversion pour convertir une référence en une valeur possédée.

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

Goutte

Lorsque le propriétaire d'une valeur disparaît, nous disons que Rust *baisse* la valeur. La suppression d'une valeur implique la libération des autres valeurs, du stockage en tas et des ressources système que la valeur possède. Les baisses se produisent dans diverses circonstances : lorsqu'une variable sort de sa portée; à la fin d'une instruction d'expression ; lorsque vous tronquez un vecteur, en supprimant des éléments de son extrémité ; et ainsi de suite.

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>
}
```

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

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

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

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

```
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 la seconde à , la première est abandonnée, et lorsque nous quittons la portée de , la seconde est abandonnée. Ce code imprime les éléments suivants : `Appellation a a`

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

Puisque notre implémentation pour ne fait qu'imprimer un message, comment, exactement, sa mémoire est-elle nettoyée ? Le type implémente , en supprimant chacun de ses éléments, puis en libérant le tampon alloué au tas qu'ils occupaient. A utilise un interne pour tenir son texte, donc pas besoin de s'implémenter lui-même; il lui permet de s'occuper de libérer les personnages. Le même principe s'étend aux valeurs : lorsque l'on est lâché, c'est finalement l'implémentation de qui se charge en fait de libérer le contenu de chacune des chaînes, et enfin de libérer le tampon contenant les éléments du vecteur. Quant à la mémoire qui détient la valeur elle-même, elle a aussi un propriétaire, peut-être une variable locale ou une structure de données, qui est responsable de la libérer. `std::ops::Drop Appellation Vec Drop String Vec<u8> String Drop Vec Appellation Vec Drop Appellation`

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

Ce principe s'applique même lorsqu'une variable peut ou non avoir vu sa valeur déplacée, en fonction du flux de contrôle. Dans des cas comme celui-ci, 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 que l'autre retourne ou , soit ou finira par posséder le , avec l'autre non initialisé. L'endroit où il atterrit détermine s'il est abandonné avant ou après le , puisqu'il sort du champ d'application avant le , et après. Bien qu'une valeur puisse être déplacée d'un endroit à l'autre, Rust ne la laisse tomber qu'une seule
fois. `complicated_condition true false p q Appellation printl n! q println! p`

Vous n'aurez généralement pas besoin d'implémenter à 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 : `std::ops::Drop`

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

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

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

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

Si un type implémente , il ne peut pas implémenter le trait. Si un type est , 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 méthode plus d'une fois sur les mêmes données. `Drop Copy Copy drop`

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

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

En d'autres termes, il reçoit son argument par valeur, prenant la propriété de l'appelant, puis n'en fait rien. La rouille diminue la valeur du moment où elle sort de son champ d'application, comme elle le ferait pour toute autre variable. `_x`

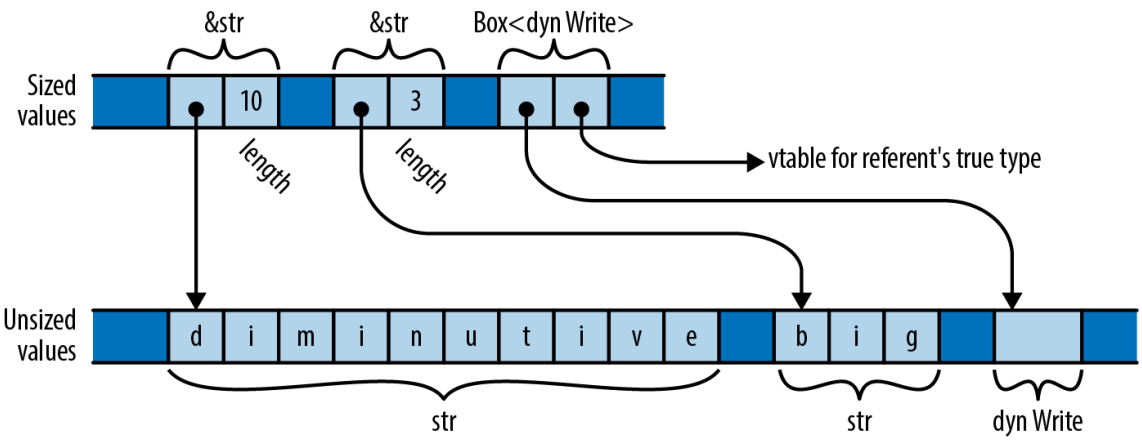
Taille

Un *type de taille* est un type dont les valeurs ont toutes la même taille en mémoire. Presque tous les types de Rust sont dimensionnés: chaque prend huit octets, chaque tuple douze. Même les enums sont dimensionnés : quelle que soit la variante réellement présente, un enum occupe toujours suffisamment d'espace pour contenir sa plus grande variante. Et bien qu'un possède un tampon alloué au tas dont la taille peut varier, la valeur elle-même est un pointeur vers le tampon, sa capacité et sa longueur, de même qu'un type de taille. `u64 (f32, f32, f32) Vec<T> Vec Vec<T>`

Tous les types de taille implémentent le 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 le mettre en œuvre vous-même. La seule utilisation pour est comme une limite pour les variables de type: un comme lié nécessite 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 d'intérêt. `std::marker::Sized Sized T: Sized T`

Cependant, Rust a également *quelques types non dimensionnés* dont les valeurs ne sont pas toutes de la même taille. Par exemple, le type de tranche de chaîne (note, sans) n'est pas dimensionné. Les littéraux de chaîne et sont des références à des tranches qui occupent dix et trois octets. Les deux sont illustrés à la figure 13-1. Les types de tranches de tableau comme (encore une fois, sans) sont également non dimensionnés : une référence partagée peut pointer vers une tranche de n'importe quelle taille. Étant donné que les et types désignent des ensembles de valeurs de tailles variables, il s'agit de types non

dimensionnés. `str & "diminutive" "big" str [T] & &[u8] [u8] str [T]`



Graphique 13-1. Références à des valeurs non dimensionnées

L'autre type commun de type non dimensionné dans Rust est un 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 et sont des pointeurs vers une valeur qui implémente le trait. Le référent peut être un fichier ou un socket réseau ou un type de votre propre pour lequel vous avez implémenté . Étant donné que l'ensemble des types qui implémentent est ouvert, considéré comme un type est non dimensionné: ses valeurs ont différentes tailles. `dyn &dyn std::io::Write Box<dyn std::io::Write> Write Write Write dyn Write`

Rust ne peut pas stocker des valeurs non dimensionnées dans des variables ou les transmettre comme arguments. Vous ne pouvez les traiter qu'à travers des pointeurs comme ou , qui sont 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 un vtable d'implémentations de méthode. `&str Box<dyn Write>`

Les objets traits et les pointeurs vers les tranches sont bien symétriques. Dans les deux cas, le type manque d'informations nécessaires pour l'utiliser : vous ne pouvez pas indexer a sans connaître sa longueur, ni appeler une méthode sur a sans connaître l'implémentation de approprié à la valeur spécifique à laquelle il se réfère. Et dans les deux cas, le pointeur de graisse remplit les informations manquantes dans le type, portant un pointeur de longueur ou de vtable. Les informations statiques omises

sont remplacées par des informations dynamiques. `[u8] Box<dyn Write> Write`

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

Malgré leurs restrictions, les types non dimensionnés rendent le système de type de Rust plus fluide. En lisant la documentation standard de la bibliothèque, vous rencontrerez parfois une liaison sur une variable de type; cela signifie presque toujours que le type donné est uniquement pointé vers et permet au code associé de fonctionner avec des tranches et des objets de trait ainsi que des valeurs ordinaires. Lorsqu'une variable de type a la limite, les gens dis-le souvent est *de taille douteuse*: elle peut l'être ou non. `?Sized ?Sized Sized`

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

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

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

Le réel 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 avec des types de taille, comme ; le résultat est un type de structure de taille. Ou vous pouvez l'utiliser avec des types non dimensionnés, comme (où est le trait pour les types qui peuvent être formatés par et des macros similaires); est un type de structure non dimensionné. `RcBox RcBox RcBox<String> RcBox<dyn std::fmt::Display> Display println! RcBox<dyn Display>`

Vous ne pouvez pas créer une valeur directement. Au lieu de cela, vous devez d'abord créer un ordinaire, de taille dont le type implémente , comme . La rouille permet alors de convertir une référence en référence graisse : `RcBox<dyn Display> RcBox value Display RcBox<String> &RcBox<String> & 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 de la transmission de valeurs à des fonctions, de sorte que vous pouvez passer un à une fonction qui attend un : `&RcBox<String> &RcBox<dyn Display>`

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

display(&boxed_lunch);
```

Cela produirait les résultats suivants :

```
For your enjoyment: lunch
```

Clone

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

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

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

Le clonage d'une valeur implique généralement l'attribution de copies de tout ce qu'elle possède, de sorte qu'un peut être coûteux, à la fois en temps et en mémoire. Par exemple, le clonage a non seulement copie le vecteur, mais copie également chacun de ses éléments. C'est pourquoi Rust ne se contente pas de cloner automatiquement les valeurs, mais vous oblige plutôt à effectuer un appel de méthode explicite. Les types de pointeurs comptés par référence sont des exceptions : le clonage de l'un d'entre eux incrémente simplement le nombre de références et vous remet un nouveau
pointeur. `clone Vec<String> String Rc<T> Arc<T>`

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

Si votre implémentation s'applique simplement 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 est assez bonne, alors Rust l'implémentera pour vous: mettez simplement au-dessus de votre définition de type. `Clone clone clone_from #[derive(Clone)]`

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

Copier

Au chapitre 4, nous avons expliqué que, pour la plupart des types, l'affectation déplace les valeurs plutôt que de les copier. Le déplacement des valeurs rend beaucoup plus simple 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 types, où l'affectation fait une copie de la source, plutôt que de déplacer la valeur et de laisser la source non initialisée. `Copy`

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

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

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

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

Mais parce qu'il s'agit d'un trait marqueur ayant une signification particulière pour le langage, Rust permet à un type de l'implémenter unique-

ment si une copie superficielle octet pour octet est tout ce dont il a besoin. Les types qui possèdent d'autres ressources, telles que les tampons de tas ou les descripteurs de système d'exploitation, ne peuvent pas implémenter . Copy Copy Copy

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

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

Réfléchissez bien avant de faire un type . Bien que cela rende le type plus facile à utiliser, il 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 ». Copy

Deref et DerefMut

Vous pouvez spécifier comment les opérateurs de déréférencement aiment et se comportent sur vos types en implémentant les traits et. Les types de pointeurs aiment et 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 valeur , alors fait référence à la valeur qui pointe vers, et fait référence à sa composante réelle. Si le contexte attribue ou emprunte une référence mutable au référent, Rust utilise le trait (« déréférencer mutable ») ; sinon, l'accès en lecture seule suffit et utilise . * . std::ops::Deref std::ops::DerefMut Box<T> Rc<T> Box<Complex> b *b Complex b b.re DerefMut Deref

Les traits sont définis comme suit:

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

trait DerefMut: Deref {
```

```
    fn deref_mut(&mut self) -> &mut Self::Target;
}
```

Les méthodes et prennent une référence et renvoient une référence. devrait être quelque chose qui contient, possède ou fait référence à: pour le type est . Notez que cela s'étend : si vous pouvez déréférencer quelque chose et le modifier, vous devriez certainement pouvoir emprunter une référence partagée à celui-ci également. Étant donné que les méthodes renvoient une référence ayant la même durée de vie que , reste empruntée aussi longtemps que la référence renvoyée
vit. `deref deref_mut &Self &Self::Target Target Self Box<Complex> Target Complex DerefMut Deref &self self`

Les et traits jouent également un autre rôle. Puisque prend une référence et renvoie une référence, Rust l'utilise pour convertir automatiquement les références du premier type en seconde. En d'autres termes, si l'insertion d'un appel empêche une incompatibilité de type, Rust en insère un pour vous. L'implémentation active la conversion correspondante pour les références modifiables. C'est ce qu'on appelle les *coercitions deref* : un type est « contraint » à se comporter comme un
autre. `Deref DerefMut deref &Self &Self::Target deref DerefMut`

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

- Si vous avez une certaine valeur et que vous voulez l'appliquer, vous pouvez simplement écrire , au lieu de : l'appel de méthode emprunte implicitement , et contraint à , car implémente
  `.Rc<String> r String::find r.find('?')`
  `(*r).find('?') r &Rc<String> &String Rc<T> Deref<Target=T>`
- Vous pouvez utiliser des méthodes telles que sur les valeurs, même s'il s'agit d'une méthode du type de tranche, car implémente . Il n'est pas nécessaire de réimplémenter toutes les méthodes de , car vous pouvez contraindre un fichier
  `.split_at String split_at str String Deref<Target=str> String str &str &String`
- Si vous avez un vecteur d'octets et que vous souhaitez le passer à une fonction qui attend une tranche d'octets , vous pouvez simplement

passer comme argument, puisque implémente . `v &[u8] &v Vec<T> Deref<Target=[T]>`

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

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 que le comportement comme les revendications de commentaire de document, vous devez implémenter et pour le type : `Selector Deref DerefMut`

```
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 comme ceci: `Selector`

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

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

// Assert that 'z' is alphabetic, using a method of `char` directly or
// `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 et traits sont conçus pour implémenter des types de pointeurs intelligents, tels que , et , et des types qui servent de versions propriétaires de quelque chose que vous utiliseriez également fréquemment par référence, de la manière et servir de versions propriétaires de et . Vous ne devez pas implémenter et pour un type juste pour faire apparaître automatiquement les méthodes du type dessus, la façon dont les méthodes d'une classe de base C ++ sont visibles sur une sous-classe. Cela ne fonctionnera pas toujours comme vous vous y attendez et peut être déroutant lorsque cela tourne
mal. `Deref DerefMut Box Rc Arc Vec<T> String [T] str Deref DerefMut Target`

Les coercitions deref viennent avec une mise en garde qui peut causer une certaine 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 , Rust voit un argument de type et un paramètre de type , trouve l'implémentation et réécrit l'appel en tant que , selon les

besoins. `show_it(&s) &Selector<&str> &str Deref<Target=str> s how_it(s.deref())`

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

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

Rust 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 déroutant : comment rendre une fonction générique pourrait-elle introduire une erreur ? Certes, ne s'implémente pas lui-même, mais il fait référence à , ce qui est certainement le cas. `Selector<&str> Display &str`

Étant donné que vous passez un argument de type et que le type de paramètre de la fonction est , la variable de type doit être . Ensuite, Rust vérifie si la liaison est satisfaite : comme elle n'applique pas de contraintes de ref pour satisfaire les limites sur les variables de type, cette vérification échoue. `&Selector<&str> &T T Selector<&str> T: Display`

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

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

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

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

Faire défaut

Certains types ont une valeur par défaut raisonnablement évidente : le vecteur ou la chaîne par défaut est vide, le nombre par défaut est zéro, la valeur par défaut est , et ainsi de suite. Des types comme celui-ci peuvent implémenter le trait: `Option None std::default::Default`

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

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

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

Tous les types de collection de Rust—, , , etc. : implémentez , avec des 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 du trait divise les valeurs produites par l'itérateur en deux collections, en utilisant une fermeture pour décider où va chaque valeur : `Vec HashMap BinaryHeap Default default Iterator partition`

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

```
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 est de produire des valeurs par défaut pour les structs qui représentent une grande collection de paramètres, dont la plupart n'auront généralement pas besoin de modifier. Par exemple, la caisse fournit des liaisons Rust pour la bibliothèque graphique OpenGL puissante et complexe. La structure comprend 24 champs, chacun contrôlant un détail différent de la façon dont OpenGL doit rendre un peu de graphiques. La fonction attend une struct comme argument. Puisque implémente , vous pouvez en créer un à passer à , en mentionnant uniquement les champs que vous souhaitez
modifier: `Default glium glium::DrawParameters glium draw Draw Parameters DrawParameters Default draw`

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

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

Cela appelle à créer une valeur initialisée avec les valeurs par défaut de tous ses champs, puis utilise la syntaxe des structs pour en créer une nouvelle avec les champs et modifiés, prêt à vous permettre de le passer à `. Default::default() DrawParameters .. line_width point_size t arget.draw`

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

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

Rust n'implémente pas implicitement pour les types struct, mais si tous les champs d'une struct implémentent , vous pouvez implémenter automatiquement pour la struct à l'aide de . `Default Default Default # [derive(Default)]`

AsRef et AsMut

Lorsqu'un type implémente , cela signifie que vous pouvez en emprunter un efficacement. est l'analogue des références modifiables. Leurs définitions sont les suivantes : `AsRef<T> &T AsMut`

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

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

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

`AsRef` est généralement utilisé pour rendre les fonctions plus flexibles dans les types d'arguments qu'elles acceptent. Par exemple, la fonction est déclarée comme suit : `std::fs::File::open`

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

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

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

Heureusement, la bibliothèque standard inclut l'implémentation générale :

```
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 et , si , alors aussi bien: suivez simplement la référence et procédez comme avant. En particulier, depuis , alors aussi. Dans un sens, c'est un moyen d'obtenir une forme limitée de coercition deref dans la vérification des limites sur les variables de type. `T U T: AsRef<U> &T: AsRef<U> str: AsRef<Path> &str: AsRef<Path> AsRef`

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

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

Emprunter et EmprunterMut

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

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

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

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

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

```
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, est ; cette signature vous obligerait à passer un by value à chaque appel à , ce qui est clairement du gaspillage. Vous avez vraiment besoin d'une référence à la clé: `K String String get`

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

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

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

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

Il devrait être assez bon pour passer tout ce qui peut être haché et comparé à notre type de clé; a devrait être parfaitement adéquat, par exem-

ple. Voici donc l'itération finale, qui est ce que vous trouverez dans la bibliothèque standard: `&str`

```
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 clé et que la référence résultante hache et compare exactement la clé elle-même, alors clairement devrait être un type de clé acceptable. Puisque implémente et , cette version finale de vous permet de passer soit ou comme une clé, selon les besoins. `&Q &Q String Borrow<str> Borrow<String> get &String &str`

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

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

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

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

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

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

De et vers

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

Leurs définitions sont joliment 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 à lui-même : chaque type implémente et `. T From<T> 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 pour rendre vos fonctions plus flexibles dans les arguments qu'ils acceptent. Par exemple, si vous écrivez : `Into`

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

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

Comme dans la section précédente, l'effet est un peu similaire à celui de la surcharge d'une fonction en C++. Avec la définition d'avant, nous pouvons faire n'importe lequel de ces appels: `AsRef ping`

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

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

```
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.

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

Étant donné que les méthodes de conversion prennent possession de leurs arguments, une conversion peut réutiliser les ressources de la

valeur d'origine pour construire la valeur convertie. Par exemple, supposons que vous écriviez : `from into`

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

L'implémentation de for prend simplement le tampon de tas de 's et le réutilise, inchangé, en tant que 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 mouvements permettent des implémentations efficaces. `Into<Vec<u8>> String String`

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

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

L'opérateur utilise et aide à nettoyer le code dans les fonctions qui pourraient échouer de plusieurs façons en convertissant automatiquement des types d'erreur spécifiques en types d'erreur généraux en cas de besoin. `? From Into`

Par exemple, imaginez un système qui a besoin de lire des données binaires et d'en convertir une partie à partir de nombres de base 10 écrits

sous forme de texte UTF-8. Cela signifie utiliser et l'implémentation pour , qui peut chacun renvoyer des erreurs de différents types. En supposant que nous utilisions les types que nous avons définis au chapitre 7 lorsque nous discutons de la gestion des erreurs, l'opérateur effectuera la conversion pour
nous: `std::str::from_utf8 FromStr i32 GenericError 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'erreur, et implémentez le trait, et la bibliothèque standard nous donne une couverture pour convertir de tout ce qui implémente en un , qui utilise
automatiquement: `Utf8Error ParseIntError Error From impl Error Box<dyn Error> ?`

```
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 grande avec deux instructions en une seule ligne. `match`

Avant et ont été ajoutés à la bibliothèque standard, le code Rust était plein de traits de conversion ad hoc et de méthodes de construction, chacun spécifique à un seul type. et 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 traits comme moyen canonique et standardisé d'encoder les conversions. `From Into From Into`

`From` et sont des traits infaillibles : leur API exige que les conversions n'échouent pas. Malheureusement, de nombreuses conversions sont plus complexes que cela. Par exemple, les grands entiers comme peuvent

stocker des nombres beaucoup plus grands que , et convertir un nombre comme en n'a pas beaucoup de sens sans quelques informations supplémentaires. Faire une simple conversion binaire, dans laquelle les 32 premiers bits sont jetés, ne donne pas souvent le résultat que nous espérions: `Into i64 i32 2_000_000_000_000i64 i32`

```
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 « wrapping » peut être appropriée. D'autre part, des applications telles que le traitement du signal numérique 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 ce maximum.

TryFrom et TryInto

Comme il n'est pas clair comment une telle conversion devrait se comporter, Rust n'implémente pas pour , ou toute autre conversion entre types numériques qui perdrait des informations. Au lieu de cela, implémente . et sont les cousins faillibles de et sont réciproques de la même manière; la mise en œuvre signifie que cela est également mis en œuvre. `From<i64> i32 i32 TryFrom<i64> TryFrom TryInto From Into TryFrom TryInto`

Leurs définitions ne sont qu'un peu plus complexes que et . `From 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 méthode nous donne un , afin que nous puissions choisir quoi faire dans le cas exceptionnel, comme un nombre trop grand pour tenir dans

le type résultant: `try_into() Result`

```
// 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 méthode de : `unwrap_or_else() 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 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 au-delà du fait qu'une erreur s'est produite, car la seule erreur possible est un débordement. D'autre part, les conversions entre types plus complexes peuvent vouloir renvoyer plus d'informations : `Error`

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

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

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

ToOwned

Étant donné une référence, la façon habituelle de produire une copie propre de son référent est d'appeler , en supposant que le type implémente . Mais que se passe-t-il si vous voulez cloner un ou un ? Ce que vous voulez probablement, c'est un ou un , mais la définition de ne le permet pas : par définition, le clonage d'un doit toujours renvoyer une valeur de type , et sont non dimensionnés ; ce ne sont même pas des types qu'une fonction pourrait renvoyer. `clone std::clone::Clone &str &
[i32] String Vec<i32> Clone &T T str [u8]`

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

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

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

Emprunter et posséder au travail : l'humble vache

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 l'une ou l'autre approche, et le type du paramètre reflète votre décision. Mais dans certains cas, vous ne pouvez pas décider d'emprunter ou de posséder tant que le programme n'est pas en cours d'exécution; le type (pour « clone on write ») fournit un moyen de le faire. `std::borrow::Cow`

Sa définition est présentée ici :

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

A emprunte une référence partagée à a ou possède une valeur à partir de laquelle nous pourrions emprunter une telle référence. Puisque implémente , vous pouvez appeler des méthodes dessus comme s'il s'agissait d'une référence partagée à un : si c'est , il emprunte une référence partagée à la valeur possédée ; et si c'est , il distribue simplement la référence qu'il détient. `Cow<B> B Cow Deref B Owned Borrowed`

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

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

Une utilisation courante consiste à renvoyer une constante de chaîne allouée statiquement ou une chaîne calculée. Par exemple, supposons que vous deviez convertir un énumérateur 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 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 l'implémentation de de pour construire les valeurs. La plupart des bras de cette instruction renvoient une référence à une chaîne allouée statiquement. Mais lorsque nous obtenons une variante, nous utilisons pour construire un message incorporant le nom de fichier donné. Ce bras de l'instruction produit une valeur. `Cow Into match Cow::Borrowed FileNotFound format! match Cow::Owned`

Les appelants qui n'ont pas besoin de changer la valeur peuvent simplement traiter le comme un : `describe Cow &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 aides et de ses appelants reporte l'allocation jusqu'au moment où elle devient nécessaire. `Cow describe`