Chapitre 11. Traits et génériques

> *[Un] informaticien a tendance à être capable de traiter des structures non uniformes – cas 1, cas 2, cas 3 – tandis qu'un mathématicien aura tendance à vouloir un axiome unificateur qui régit un système entier.*
>
> —Donald Knuth

L'une des grandes découvertes de la programmation est qu'il est possible d'écrire du code qui fonctionne sur des valeurs de nombreux types différents, *même des types qui n'ont pas encore été inventés.* Voici deux exemples :

- `Vec<T>` est générique : vous pouvez créer un vecteur de n'importe quel type de valeur, y compris des types définis dans votre programme que les auteurs n'ont jamais anticipés. `Vec`
- Beaucoup de choses ont des méthodes, y compris s et s. Votre code peut prendre un scripteur par référence, n'importe quel scripteur, et lui envoyer des données. Votre code n'a pas à se soucier du type d'écriture dont il s'agit. Plus tard, si quelqu'un ajoute un nouveau type de graveur, votre code le prendra déjà en charge. `.write() File TcpStream`

Bien sûr, cette capacité n'est pas nouvelle avec Rust. C'est ce qu'on appelle le *polymorphisme*, et c'était la nouvelle technologie de langage de programmation des années 1970. À l'heure actuelle, il est effectivement universel. Rust supporte le polymorphisme avec deux caractéristiques connexes: les traits et les génériques. Ces concepts seront familiers à de nombreux programmeurs, mais Rust adopte une nouvelle approche inspirée des classes de type de Haskell.

Les traits sont la vision de Rust sur les interfaces ou les classes de base abstraites. Au début, ils ressemblent à des interfaces en Java ou C#. Le trait d'écriture d'octets est appelé , et sa définition dans la bibliothèque standard commence comme ceci: `std::io::Write`

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

Ce trait offre plusieurs méthodes; nous n'avons montré que les trois premiers.

Les types standard et les deux implémentent . Il en va de même pour . Les trois types fournissent des méthodes nommées , etc. Le code qui utilise un scripteur sans se soucier de son type ressemble à ceci
`:File TcpStream std::io::Write Vec<u8> .write() .flush()`

```
use std::io::Write;

fn say_hello(out: &mut dyn Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Le type de est , ce qui signifie « une référence mutable à toute valeur qui implémente le trait ». Nous pouvons passer une référence mutable à une telle valeur: `out &mut dyn Write Write say_hello`

```
use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;  // works

let mut bytes = vec![];
say_hello(&mut bytes)?;  // also works
assert_eq!(bytes, b"hello world\n");
```

Ce chapitre commence par montrer comment les traits sont utilisés, comment ils fonctionnent et comment définir les vôtres. Mais il y a plus de traits que ce que nous avons laissé entendre jusqu'à présent. Nous les utiliserons pour ajouter des méthodes d'extension aux types existants, même des types intégrés comme et . Nous expliquerons pourquoi l'ajout d'un trait à un type ne coûte pas de mémoire supplémentaire et comment utiliser les traits sans surcharge d'appel de méthode virtuelle. Nous verrons que les traits intégrés sont le crochet dans le langage que Rust fournit pour la surcharge de l'opérateur et d'autres fonctionnalités. Et nous couvrirons le type, les fonctions associées et les types associés, trois fonctionnalités que Rust a retirées de Haskell et qui résolvent élégamment les problèmes que d'autres langages abordent avec des solutions de contournement et des hacks. `str bool Self`

Les génériques sont l'autre saveur du polymorphisme dans Rust. Comme un modèle C++, une fonction ou un type générique peut être utilisé avec des valeurs de nombreux types différents :

```
/// Given two values, pick whichever one is less.
fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}
```

La fonction dans cette fonction signifie qu'elle peut être utilisée avec des arguments de n'importe quel type qui implémente le trait, c'est-à-dire n'importe quel type ordonné. Une exigence comme celle-ci est appelée une *limite*, car elle fixe des limites sur les types qui pourraient éventuellement l'être. Le compilateur génère du code machine personnalisé pour chaque type que vous utilisez réellement. `<T: Ord> min T Ord T T`

Les génériques et les traits sont étroitement liés : les fonctions génériques utilisent des traits dans les limites pour énoncer les types d'arguments auxquels elles peuvent être appliquées. Nous parlerons donc également de la façon dont et sont similaires, en quoi ils sont différents et comment choisir entre ces deux façons d'utiliser les traits. `&mut dyn Write <T: Write>`

Utilisation des traits

Un trait est une caractéristique qu'un type donné peut ou non prendre en charge. Le plus souvent, un trait représente une capacité : quelque chose qu'un type peut faire.

- Une valeur qui implémente peut écrire des octets. `std::io::Write`
- Une valeur qui implémente peut produire une séquence de valeurs. `std::iter::Iterator`
- Une valeur qui implémente peut faire des clones d'elle-même en mémoire. `std::clone::Clone`
- Une valeur qui implémente peut être imprimée à l'aide du spécificateur de format. `std::fmt::Debug println!() {:?}`

Ces quatre traits font tous partie de la bibliothèque standard de Rust, et de nombreux types standard les implémentent. Par exemple:

- `std::fs::File` met en œuvre le trait; il écrit des octets dans un fichier local. écrit dans une connexion réseau. implémente également .

Chaque appel sur un vecteur d'octets ajoute des données à la fin. `Write` `std::net::TcpStream` `Vec<u8>` `Write` `.write()`

- `Range<i32>` (le type de) implémente le trait, tout comme certains types d'itérateurs associés aux tranches, aux tables de hachage, etc. `0..10` `Iterator`
- La plupart des types de bibliothèque standard implémentent . Les exceptions sont principalement des types comme celui-ci qui représentent plus que de simples données en mémoire. `Clone` `TcpStream`
- De même, la plupart des types de bibliothèque standard prennent en charge . `Debug`

Il existe une règle inhabituelle à propos des méthodes de trait: le trait lui-même doit être dans la portée. Sinon, toutes ses méthodes sont cachées:

```
let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?;  // error: no method named `write_all`
```

Dans ce cas, le compilateur imprime un message d'erreur convivial qui suggère d'ajouter et en effet qui résout le problème: `use std::io::Write;`

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?;  // ok
```

Rust a cette règle car, comme nous le verrons plus loin dans ce chapitre, vous pouvez utiliser des traits pour ajouter de nouvelles méthodes à n'importe quel type, même les types de bibliothèque standard comme et . Les caisses tierces peuvent faire la même chose. De toute évidence, cela pourrait conduire à des conflits de noms! Mais puisque Rust vous fait importer les traits que vous prévoyez d'utiliser, les caisses sont libres de profiter de ce superpouvoir. Pour obtenir un conflit, vous devez importer deux traits qui ajoutent une méthode portant le même nom au même type. C'est rare dans la pratique. (Si vous rencontrez un conflit, vous pouvez énoncer ce que vous voulez à l'aide d'une syntaxe de méthode complète, abordée plus loin dans le chapitre.) `u32` `str`

La raison et les méthodes fonctionnent sans aucune importation spéciale est qu'elles sont toujours dans le champ d'application par défaut : elles font partie du prélude standard, des noms que Rust importe automatiquement dans chaque module. En fait, le prélude est principalement une

sélection soigneusement choisie de traits. Nous couvrirons beaucoup d'entre eux dans le chapitre 13. `Clone Iterator`

Les programmeurs C++ et C# auront déjà remarqué que les méthodes de trait sont comme des méthodes virtuelles. Pourtant, les appels comme celui montré ci-dessus sont rapides, aussi rapides que n'importe quel autre appel de méthode. En termes simples, il n'y a pas de polymorphisme ici. Il est évident qu'il s'agit d'un vecteur, pas d'un fichier ou d'une connexion réseau. Le compilateur peut émettre un simple appel à . Il peut même intégrer la méthode. (C++ et C# feront souvent de même, bien que la possibilité de sous-classification l'empêche parfois.) Seuls les appels via encourent la surcharge d'une répartition dynamique, également appelée appel de méthode virtuelle, qui est indiquée par le mot-clé dans le type. est connu comme un *objet trait*; nous examinerons les détails techniques des objets traits et leur comparaison avec les fonctions génériques dans les sections suivantes. `buf Vec<u8>::write() &mut dyn Write dyn dyn Write`

Objets Trait

Il existe deux façons d'utiliser des traits pour écrire du code polymorphe dans Rust : les objets de trait et les génériques. Nous présenterons d'abord les objets de trait et nous nous tournerons vers les génériques dans la section suivante.

La rouille n'autorise pas les variables de type : `dyn Write`

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: dyn Write = buf;  // error: `Write` does not have a constant
```

La taille d'une variable doit être connue au moment de la compilation, et les types qui implémentent peuvent être de n'importe quelle taille. `Write`

Cela peut surprendre si vous venez de C# ou de Java, mais la raison est simple. En Java, une variable de type (l'interface standard Java analogue à) est une référence à tout objet qui implémente . Le fait qu'il s'agisse d'une référence va de soi. C'est la même chose avec les interfaces en C# et dans la plupart des autres
langages. `OutputStream std::io::Write OutputStream`

Ce que nous voulons dans Rust, c'est la même chose, mais dans Rust, les références sont explicites :

```
let mut buf: Vec<u8> = vec![];
let writer: &mut dyn Write = &mut buf;  // ok
```

Une référence à un type de trait, comme , est appelée un *objet de trait*. Comme toute autre référence, un objet trait pointe vers une valeur, il a une durée de vie et il peut être partagé ou partagé. `writer mut`

Ce qui rend un objet trait différent, c'est que Rust ne connaît généralement pas le type du référent au moment de la compilation. Ainsi, un objet trait comprend un peu plus d'informations sur le type du référent. Ceci est strictement pour l'usage propre de Rust dans les coulisses: lorsque vous appelez , Rust a besoin des informations de type pour appeler dynamiquement la bonne méthode en fonction du type de . Vous ne pouvez pas interroger directement les informations de type, et Rust ne prend pas en charge le downcasting de l'objet trait vers un type concret comme `.writer.write(data) write *writer &mut dyn Write Vec<u8>`

Disposition de l'objet Trait

En mémoire, un objet trait est un pointeur gras composé d'un pointeur vers la valeur, plus un pointeur vers une table représentant le type de cette valeur. Chaque objet trait prend donc deux mots machine, comme le montre la figure 11-1.

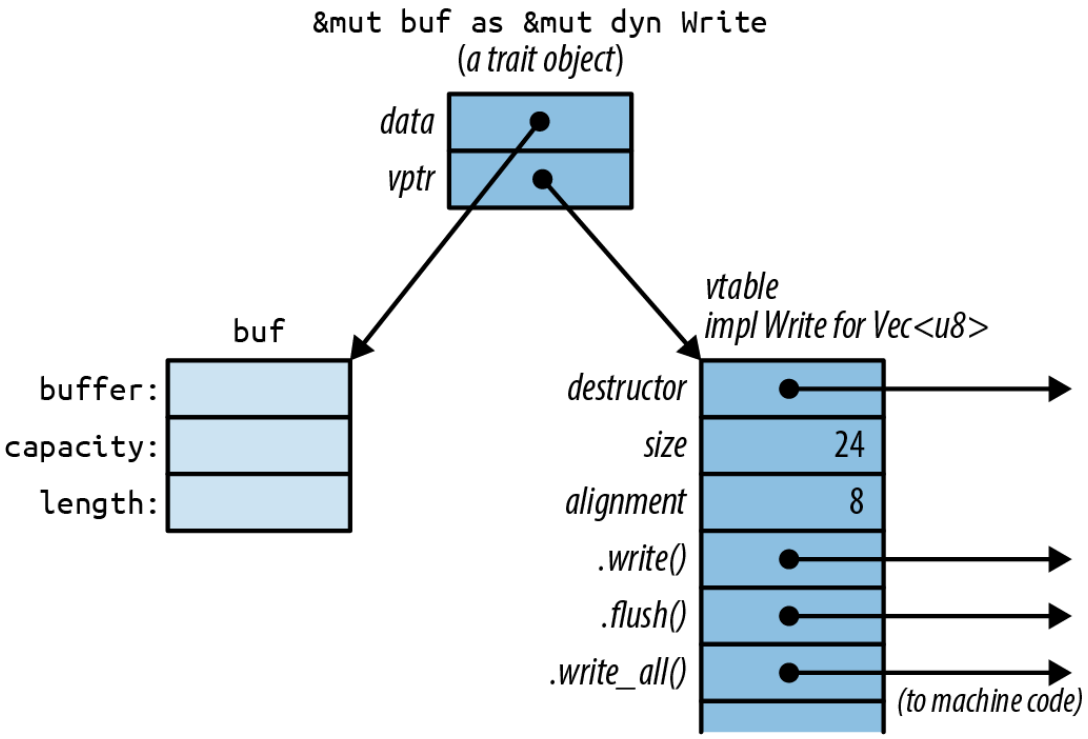


Figure 11-1. Objets traits en mémoire

C++ contient également ce type d'informations de type d'exécution. C'est ce qu'on appelle une *table virtuelle*, ou *vtable*. Dans Rust, comme en C++, le vtable est généré une fois, au moment de la compilation, et partagé par

tous les objets du même type. Tout ce qui est montré dans la teinte la plus sombre de la figure 11-1, y compris le vtable, est un détail d'implémentation privé de Rust. Encore une fois, ce ne sont pas des champs et des structures de données auxquels vous pouvez accéder directement. Au lieu de cela, le langage utilise automatiquement le vtable lorsque vous appelez une méthode d'un objet trait pour déterminer l'implémentation à appeler.

Les programmeurs C++ chevronnés remarqueront que Rust et C++ utilisent la mémoire un peu différemment. En C++, le pointeur vtable, ou *vptr*, est stocké dans le cadre de la structure. Rust utilise plutôt des pointeurs de graisse. La structure elle-même ne contient rien d'autre que ses champs. De cette façon, une structure peut implémenter des dizaines de traits sans contenir des dizaines de vptrs. Même des types comme , qui ne sont pas assez grands pour accueillir un vptr, peuvent implémenter des traits. `i32`

Rust convertit automatiquement les références ordinaires en objets traits en cas de besoin. C'est pourquoi nous sommes en mesure de passer à dans cet exemple: `&mut local_file say_hello`

```
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?;
```

Le type de est , et le type de l'argument à est . Puisque a est une sorte d'écrivain, Rust le permet, convertissant automatiquement la référence simple en un objet trait. `&mut local_file &mut File say_hello &mut dyn Write File`

De même, Rust convertira volontiers a en , une valeur qui possède un écrivain dans le tas: `Box<File> Box<dyn Write>`

```
let w: Box<dyn Write> = Box::new(local_file);
```

`Box<dyn Write>` , comme , est un gros pointeur : il contient l'adresse de l'auteur lui-même et l'adresse du vtable. Il en va de même pour d'autres types de pointeurs, comme . `&mut dyn Write Rc<dyn Write>`

Ce type de conversion est le seul moyen de créer un objet trait. Ce que le compilateur fait réellement ici est très simple. Au moment où la conversion se produit, Rust connaît le vrai type du référent (dans ce cas,), il ajoute donc simplement l'adresse du vtable approprié, transformant le pointeur régulier en un gros pointeur. `File`

Fonctions génériques et paramètres de type

Au début de ce chapitre, nous avons montré une fonction qui prenait un objet trait comme argument. Réécrivons cette fonction en tant que fonction générique : `say_hello()`

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Seule la signature de type a changé :

```
fn say_hello(out: &mut dyn Write)     // plain function

fn say_hello<W: Write>(out: &mut W)   // generic function
```

La phrase est ce qui rend la fonction générique. Il s'agit d'un *paramètre de type*. Cela signifie que dans tout le corps de cette fonction, représente un type qui implémente le trait. Les paramètres de type sont généralement des lettres majuscules simples, par convention. `<W: Write> W Write`

Le type signifie dépend de la façon dont la fonction générique est utilisée: `W`

```
say_hello(&mut local_file)?;  // calls say_hello::<File>
say_hello(&mut bytes)?;       // calls say_hello::<Vec<u8>>
```

Lorsque vous passez à la fonction générique, vous appelez . Rust génère du code machine pour cette fonction qui appelle et . Lorsque vous passez , vous appelez . Rust génère un code machine distinct pour cette version de la fonction, en appelant les méthodes correspondantes. Dans les deux cas, Rust déduit le type du type de l'argument. Ce processus est connu sous le nom *de monomorphisation*, et le compilateur gère tout cela automatiquement. `&mut local_file say_hello() say_hello::<File> () File::write_all() File::flush() &mut bytes say_hello:: <Vec<u8>>() Vec<u8> W`

Vous pouvez toujours épeler les paramètres de type :

```
say_hello::<File>(&mut local_file)?;
```

Ceci est rarement nécessaire, car Rust peut généralement déduire les paramètres de type en regardant les arguments. Ici, la fonction générique attend un argument, et nous lui passons un , donc Rust en déduit que `.say_hello &mut W &mut File W = File`

Si la fonction générique que vous appelez n'a pas d'arguments qui fournissent des indices utiles, vous devrez peut-être l'épeler :

```
// calling a generic method collect<C>() that takes no arguments
let v1 = (0 .. 1000).collect();  // error: can't infer type
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

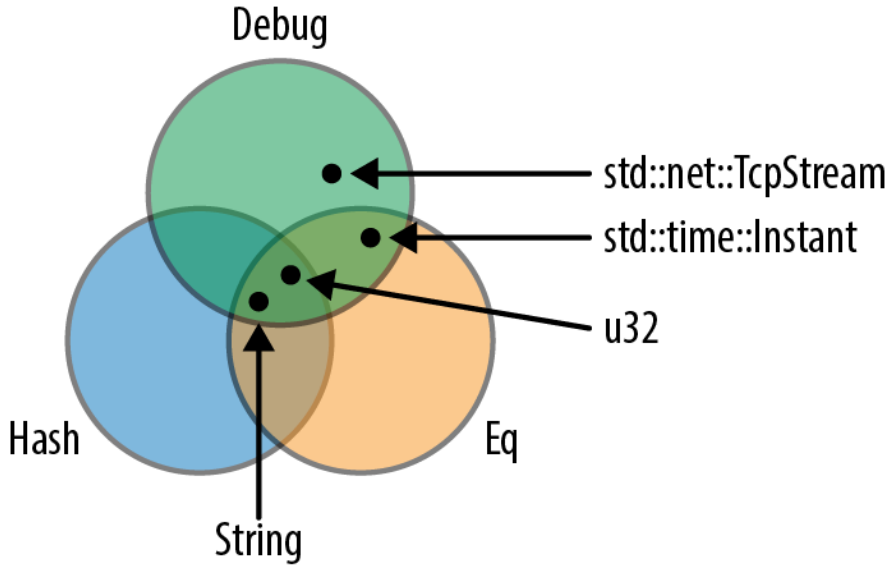
Parfois, nous avons besoin de plusieurs capacités à partir d'un paramètre de type. Par exemple, si nous voulons imprimer les dix valeurs les plus courantes dans un vecteur, nous aurons besoin que ces valeurs soient imprimables :

```
use std::fmt::Debug;

fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

Mais ce n'est pas suffisant. Comment prévoyons-nous de déterminer quelles valeurs sont les plus courantes? La méthode habituelle consiste à utiliser les valeurs comme clés dans une table de hachage. Cela signifie que les valeurs doivent soutenir les opérations. Les limites sur doivent inclure celles-ci ainsi que . La syntaxe utilise le signe : `Hash Eq T Debug +`

```
use std::hash::Hash;
use std::fmt::Debug;

fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

Certains types implémentent , certains implémentent , certains prennent en charge , et quelques-uns, comme et , implémentent les trois, comme le montre la figure 11-2. `Debug Hash Eq u32 String`



Graphique 11-2. Traits en tant qu'ensembles de types

Il est également possible qu'un paramètre de type n'ait aucune limite, mais vous ne pouvez pas faire grand-chose avec une valeur si vous n'avez spécifié aucune limite pour celle-ci. Vous pouvez le déplacer. Vous pouvez le mettre dans une boîte ou un vecteur. C'est à peu près tout.

Les fonctions génériques peuvent avoir plusieurs paramètres de type :

```
/// Run a query on a large, partitioned data set.
/// See <http://research.google.com/archive/mapreduce.html>.
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

Comme le montre cet exemple, les limites peuvent être si longues qu'elles sont dures pour les yeux. Rust fournit une syntaxe alternative en utilisant le mot-clé : `where`

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

Les paramètres de type sont toujours déclarés à l'avance, mais les limites sont déplacées vers des lignes séparées. Ce type de clause est également autorisé sur les structs génériques, les enums, les alias de type et les méthodes, partout où les limites sont autorisées. `M R where`

Bien sûr, une alternative aux clauses est de rester simple: trouver un moyen d'écrire le programme sans utiliser de génériques de manière aussi intensive. `where`

« Réception de références en tant qu'arguments de fonction » a introduit la syntaxe des paramètres de durée de vie. Une fonction générique peut avoir à la fois des paramètres de durée de vie et des paramètres de type. Les paramètres de durée de vie viennent en premier:

```
/// Return a reference to the point in `candidates` that's
/// closest to the `target` point.
fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P
    where P: MeasureDistance
{
    ...
}
```

Cette fonction prend deux arguments, et . Les deux sont des références, et nous leur donnons des durées de vie distinctes et (comme discuté dans « Paramètres de durée de vie distincts »). De plus, la fonction fonctionne avec n'importe quel type qui implémente le trait, nous pouvons donc l'utiliser sur des valeurs dans un programme et des valeurs dans un autre. `target candidates 't 'c P MeasureDistance Point2d Poin t3d`

Les durées de vie n'ont jamais d'impact sur le code machine. Deux appels à l'utilisation du même type, mais des durées de vie différentes, appelleront la même fonction compilée. Seuls des types différents amènent Rust à compiler plusieurs copies d'une fonction générique. `nearest() P`

En plus des types et des durées de vie, les fonctions génériques peuvent également prendre des paramètres constants, comme la structure que nous avons présentée dans « Structs génériques avec des paramètres constants »: `Polynomial`

```
fn dot_product<const N: usize>(a: [f64; N], b: [f64; N]) -> f64 {
    let mut sum = 0.;
    for i in 0..N {
        sum += a[i] * b[i];
    }
    sum
}
```

Ici, la phrase indique que la fonction attend un paramètre générique , qui doit être un . Étant donné , la fonction prend deux arguments de type , et additionne les produits de leurs éléments correspondants. Ce qui distingue d'un argument ordinaire, c'est que vous pouvez l'utiliser dans les

types dans la signature ou le corps de . `<const N: usize> dot_product N usize N [f64; N] N usize dot_product`

Comme pour les paramètres de type, vous pouvez soit fournir explicitement des paramètres constants, soit laisser Rust les déduire :

```
// Explicitly provide `3` as the value for `N`.
dot_product::<3>([0.2, 0.4, 0.6], [0., 0., 1.])

// Let Rust infer that `N` must be `2`.
dot_product([3., 4.], [-5., 1.])
```

Bien sûr, les fonctions ne sont pas le seul type de code générique dans Rust:

- Nous avons déjà couvert les types génériques dans « Generic Structs » et « Generic Enums ».
- Une méthode individuelle peut être générique, même si le type sur lequel elle est définie n'est pas générique :

```
impl PancakeStack {
    fn push<T: Topping>(&mut self, goop: T) -> PancakeResult<()> {
        goop.pour(&self);
        self.absorb_topping(goop)
    }
}
```

- Les alias de type peuvent également être génériques :

```
type PancakeResult<T> = Result<T, PancakeError>;
```

- Nous couvrirons les traits génériques plus loin dans ce chapitre.

Toutes les fonctionnalités présentées dans cette section (limites, clauses, paramètres de durée de vie, etc.) peuvent être utilisées sur tous les éléments génériques, pas seulement sur les fonctions. `where`

Lequel utiliser

Le choix d'utiliser des objets de trait ou du code générique est subtil. Étant donné que les deux caractéristiques sont basées sur des traits, elles ont beaucoup en commun.

Les objets traits sont le bon choix chaque fois que vous avez besoin d'une collection de valeurs de types mixtes, tous ensemble. Il est techniquement

possible de faire une salade générique:

```
trait Vegetable {
    ...
}

struct Salad<V: Vegetable> {
    veggies: Vec<V>
}
```

Cependant, il s'agit d'une conception plutôt sévère. Chacune de ces salades se compose entièrement d'un seul type de légume. Tout le monde n'est pas fait pour ce genre de chose. L'un de vos auteurs a déjà payé 14 $ pour un et n'a jamais tout à fait surmonté l'expérience. `Salad<IcebergLettuce>`

Comment pouvons-nous construire une meilleure salade? Étant donné que les valeurs peuvent être toutes de tailles différentes, nous ne pouvons pas demander à Rust un : `Vegetable Vec<dyn Vegetable>`

```
struct Salad {
    veggies: Vec<dyn Vegetable>  // error: `dyn Vegetable` does
                                 // not have a constant size
}
```

Les objets traits sont la solution :

```
struct Salad {
    veggies: Vec<Box<dyn Vegetable>>
}
```

Chacun peut posséder n'importe quel type de légume, mais la boîte elle-même a une taille constante – deux pointeurs – adaptée au stockage dans un vecteur. Mis à part la métaphore malheureuse et mixte d'avoir des boîtes dans sa nourriture, c'est précisément ce qui est nécessaire, et cela fonctionnerait tout aussi bien pour les formes dans une application de dessin, les monstres dans un jeu, les algorithmes de routage enfichables dans un routeur réseau, etc. `Box<dyn Vegetable>`

Une autre raison possible d'utiliser des objets traits est de réduire la quantité totale de code compilé. Rust peut avoir à compiler une fonction générique plusieurs fois, une fois pour chaque type avec lequel elle est utilisée. Cela pourrait rendre le binaire grand, un phénomène appelé *gonflement du code* dans les cercles C ++. De nos jours, la mémoire est abon-

dante et la plupart d'entre nous ont le luxe d'ignorer la taille du code; mais il existe des environnements contraints.

En dehors des situations impliquant des environnements à salade ou à faibles ressources, les génériques présentent trois avantages importants par rapport aux objets traits, de sorte que dans Rust, les génériques sont le choix le plus courant.

Le premier avantage est la vitesse. Notez l'absence du mot-clé dans les signatures de fonction génériques. Étant donné que vous spécifiez les types au moment de la compilation, explicitement ou par inférence de type, le compilateur sait exactement quelle méthode appeler. Le mot-clé n'est pas utilisé car il n'y a pas d'objets traits , et donc pas de répartition dynamique , impliqués. `dyn write dyn`

La fonction générique montrée dans l'introduction est aussi rapide que si nous avions écrit des fonctions séparées , , , et ainsi de suite. Le compilateur peut l'intégrer, comme n'importe quelle autre fonction, donc dans une version release, un appel à n'est probablement que deux ou trois instructions. Un appel avec des arguments constants, comme , sera encore plus rapide: Rust peut l'évaluer au moment de la compilation, de sorte qu'il n'y a aucun coût d'exécution. `min() min_u8 min_i64 min_string min::<i32> min(5, 3)`

Ou considérez cet appel de fonction générique :

```
let mut sink = std::io::sink();
say_hello(&mut sink)?;
```

`std::io::sink()` renvoie un graveur de type qui ignore discrètement tous les octets qui y sont écrits. `Sink`

Lorsque Rust génère du code machine pour cela, il peut émettre du code qui appelle , vérifie les erreurs, puis appelle . C'est ce que le corps de la fonction générique dit de faire. `Sink::write_all Sink::flush`

Ou, Rust pourrait examiner ces méthodes et réaliser ce qui suit:

- `Sink::write_all()` ne fait rien.
- `Sink::flush()` ne fait rien.
- Aucune des deux méthodes ne renvoie jamais d'erreur.

En bref, Rust dispose de toutes les informations dont il a besoin pour optimiser complètement cet appel de fonction.

Comparez cela au comportement avec des objets traits. Rust ne sait jamais vers quel type de valeur pointe un objet trait jusqu'au moment de l'exécution. Ainsi, même si vous passez un , la surcharge d'appel de méthodes virtuelles et de vérification des erreurs s'applique toujours. `Sink`

Le deuxième avantage des génériques est que tous les traits ne peuvent pas prendre en charge des objets de traits. Les traits prennent en charge plusieurs fonctionnalités, telles que les fonctions associées, qui ne fonctionnent qu'avec des génériques: ils excluent complètement les objets de trait. Nous soulignerons ces caractéristiques au fur et à mesure que nous y arriverons.

Le troisième avantage des génériques est qu'il est facile de lier un paramètre de type générique avec plusieurs traits à la fois, comme notre fonction l'a fait lorsqu'elle a eu besoin de son paramètre pour implémenter . Les objets Trait ne peuvent pas faire cela : des types comme ceux-ci ne sont pas pris en charge dans Rust. (Vous pouvez contourner ce problème avec des sous-traits, définis plus loin dans ce chapitre, mais c'est un peu impliqué.) `top_ten T Debug + Hash + Eq &mut (dyn Debug + Hash + Eq)`

Définition et mise en œuvre des traits

Définir un trait est simple. Donnez-lui un nom et listez les signatures de type des méthodes de trait. Si nous écrivons un jeu, nous pourrions avoir un trait comme celui-ci:

```
/// A trait for characters, items, and scenery -
/// anything in the game world that's visible on screen.
trait Visible {
    /// Render this object on the given canvas.
    fn draw(&self, canvas: &mut Canvas);

    /// Return true if clicking at (x, y) should
    /// select this object.
    fn hit_test(&self, x: i32, y: i32) -> bool;
}
```

Pour implémenter un trait, utilisez la syntaxe : `impl TraitName for Type`

```
impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.y - self.height - 1 .. self.y {
```

```
                canvas.write_at(self.x, y, '|');
            }
            canvas.write_at(self.x, self.y, 'M');
        }

        fn hit_test(&self, x: i32, y: i32) -> bool {
            self.x == x
            && self.y - self.height - 1 <= y
            && y <= self.y
        }
    }
```

Notez que cela contient une implémentation pour chaque méthode du trait, et rien d'autre. Tout ce qui est défini dans un trait doit en fait être une caractéristique du trait; si nous voulions ajouter une méthode d'assistance à l'appui de , nous devrions la définir dans un bloc séparé: `impl Visible impl Broom::draw() impl`

```
    impl Broom {
        /// Helper function used by Broom::draw() below.
        fn broomstick_range(&self) -> Range<i32> {
            self.y - self.height - 1 .. self.y
        }
    }
```

Ces fonctions d'assistance peuvent être utilisées dans les blocs de traits : `impl`

```
    impl Visible for Broom {
        fn draw(&self, canvas: &mut Canvas) {
            for y in self.broomstick_range() {
                ...
            }
            ...
        }
        ...
    }
```

Méthodes par défaut

Le type d'enregistreur dont nous avons parlé précédemment peut être implémenté en quelques lignes de code. Tout d'abord, nous définissons le type: `Sink`

```
/// A Writer that ignores whatever data you write to it.
pub struct Sink;
```

`Sink` est une structure vide, car nous n'avons pas besoin d'y stocker de données. Ensuite, nous fournissons une implémentation du trait pour `:Write Sink`

```
use std::io::{Write, Result};

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        // Claim to have successfully written the whole buffer.
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

Jusqu'à présent, cela ressemble beaucoup au trait. Mais nous avons également vu que le trait a une méthode: `Visible Write write_all`

```
let mut out = Sink;
out.write_all(b"hello world\n")?;
```

Pourquoi Rust nous laisse-t-il sans définir cette méthode ? La réponse est que la définition du trait de la bibliothèque standard contient une *implémentation par défaut* pour : `impl Write for Sink Write write_all`

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> {
        let mut bytes_written = 0;
        while bytes_written < buf.len() {
            bytes_written += self.write(&buf[bytes_written..])?;
        }
        Ok(())
    }

    ...
}
```

Les méthodes et sont les méthodes de base que chaque écrivain doit mettre en œuvre. Un rédacteur peut également implémenter , mais sinon, l'implémentation par défaut indiquée précédemment sera utilisée. `write flush write_all`

Vos propres caractéristiques peuvent inclure des implémentations par défaut utilisant la même syntaxe.

L'utilisation la plus spectaculaire des méthodes par défaut dans la bibliothèque standard est le trait, qui a une méthode requise () et des dizaines de méthodes par défaut. Le chapitre 15 explique pourquoi. `Iterator .next()`

Traits et types d'autres personnes

Rust vous permet d'implémenter n'importe quel trait sur n'importe quel type, tant que le trait ou le type est introduit dans la caisse actuelle.

Cela signifie que chaque fois que vous souhaitez ajouter une méthode à n'importe quel type, vous pouvez utiliser un trait pour le faire:

```
trait IsEmoji {
    fn is_emoji(&self) -> bool;
}

/// Implement IsEmoji for the built-in character type.
impl IsEmoji for char {
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

Comme toute autre méthode de trait, cette nouvelle méthode n'est visible que lorsqu'elle est dans la portée. `is_emoji IsEmoji`

Le seul but de ce trait particulier est d'ajouter une méthode à un type existant, . C'est ce qu'on appelle un *trait d'extension*. Bien sûr, vous pouvez également ajouter ce trait aux types en écrivant et ainsi de suite. `char impl IsEmoji for str { ... }`

Vous pouvez même utiliser un bloc générique pour ajouter un trait d'extension à toute une famille de types à la fois. Ce trait pourrait être mis en œuvre sur n'importe quel type: `impl`

```
use std::io::{self, Write};

/// Trait for values to which you can send HTML.
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()>;
}
```

L'implémentation du trait pour tous les écrivains en fait un trait d'extension, ajoutant une méthode à tous les écrivains Rust:

```
/// You can write HTML to any std::io writer.
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}
```

La ligne signifie « pour chaque type qui implémente , voici une implémentation de pour » `impl<W: Write> WriteHtml for W W Write WriteHtml W`

La bibliothèque offre un bel exemple de l'utilité d'implémenter des traits définis par l'utilisateur sur des types standard. est une bibliothèque de sérialisation. Autrement dit, vous pouvez l'utiliser pour écrire des structures de données Rust sur le disque et les recharger ultérieurement. La bibliothèque définit un trait, , qui est implémenté pour chaque type de données pris en charge par la bibliothèque. Donc, dans le code source, il y a du code implémentant pour , , , , les types de tableau et de tuple, et ainsi de suite, à travers toutes les structures de données standard comme et `. serde serde Serialize serde Serialize bool i8 i16 i32 Vec H ashMap`

Le résultat de tout cela est qu'il ajoute une méthode à tous ces types. Il peut être utilisé comme ceci: `serde .serialize()`

```
use serde::Serialize;
use serde_json;

pub fn save_configuration(config: &HashMap<String, String>)
    -> std::io::Result<()>
{
    // Create a JSON serializer to write the data to a file.
    let writer = File::create(config_filename())?;
    let mut serializer = serde_json::Serializer::new(writer);
```

```
        // The serde `.serialize()` method does the rest.
        config.serialize(&mut serializer)?;

        Ok(())
    }
```

Nous avons dit plus tôt que lorsque vous implémentez un trait, le trait ou le type doit être nouveau dans la caisse actuelle. C'est ce qu'on appelle la *règle orpheline*. Il aide Rust à s'assurer que les implémentations de traits sont uniques. Votre code ne peut pas , car les deux et sont définis dans la bibliothèque standard. Si Rust laissait les caisses faire cela, il pourrait y avoir plusieurs implémentations de pour , dans différentes caisses, et Rust n'aurait aucun moyen raisonnable de décider quelle implémentation utiliser pour un appel de méthode donné. `impl Write for u8 Write u8 Write u8`

(C++ a une restriction d'unicité similaire : la règle de définition unique. En C++ typique, il n'est pas appliqué par le compilateur, sauf dans les cas les plus simples, et vous obtenez un comportement indéfini si vous le cassez.)

Soi dans les traits

Un trait peut utiliser le mot-clé comme type. Le trait standard, par exemple, ressemble à ceci (légèrement simplifié): `Self Clone`

```
    pub trait Clone {
        fn clone(&self) -> Self;
        ...
    }
```

L'utilisation comme type de retour ici signifie que le type de est le même que le type de , quel qu'il soit. Si est un , alors le type de est —pas ou tout autre type
clonable. `Self x.clone() x x String x.clone() String dyn Clone`

De même, si nous définissons ce trait :

```
    pub trait Spliceable {
        fn splice(&self, other: &Self) -> Self;
    }
```

avec deux implémentations :

```
impl Spliceable for CherryTree {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}
```

puis à l'intérieur du premier , est simplement un alias pour , et dans le second, c'est un alias pour . Cela signifie que nous pouvons épisser ensemble deux cerisiers ou deux mammouths, pas que nous pouvons créer un hybride mammouth-cerise. Le type et le type de doivent correspondre. `impl Self CherryTree Mammoth self other`

Un trait qui utilise le type est incompatible avec les objets trait : `Self`

```
// error: the trait `Spliceable` cannot be made into an object
fn splice_anything(left: &dyn Spliceable, right: &dyn Spliceable) {
    let combo = left.splice(right);
    // ...
}
```

La raison en est quelque chose que nous verrons encore et encore alors que nous creusons dans les fonctionnalités avancées des traits. Rust rejette ce code car il n'a aucun moyen de taper-vérifier l'appel . Tout l'intérêt des objets traits est que le type n'est pas connu avant l'exécution. Rust n'a aucun moyen de savoir au moment de la compilation si et sera le même type, selon les besoins. `left.splice(right) left right`

Les objets Trait sont vraiment destinés aux types de traits les plus simples, les types qui pourraient être implémentés à l'aide d'interfaces en Java ou de classes de base abstraites en C++. Les fonctionnalités plus avancées des traits sont utiles, mais elles ne peuvent pas coexister avec les objets traits car avec les objets traits, vous perdez les informations de type dont Rust a besoin pour vérifier votre programme.

Maintenant, si nous avions voulu un épissage génétiquement improbable, nous aurions pu concevoir un trait respectueux des objets:

```
pub trait MegaSpliceable {
    fn splice(&self, other: &dyn MegaSpliceable) -> Box<dyn MegaSpliceab
}
```

Ce trait est compatible avec les objets traits. Il n'y a pas de problème de vérification de type des appels à cette méthode car le type de l'argument n'est pas nécessaire pour correspondre au type de , tant que les deux types sont . `.splice()` `other` `self` `MegaSpliceable`

Sous-traits

Nous pouvons déclarer qu'un trait est une extension d'un autre trait :

```
/// Someone in the game world, either the player or some other
/// pixie, gargoyle, squirrel, ogre, etc.
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}
```

La phrase signifie que toutes les créatures sont visibles. Chaque type qui implémente doit également implémenter le trait : `trait Creature: Visible` `Creature` `Visible`

```
impl Visible for Broom {
    ...
}

impl Creature for Broom {
    ...
}
```

Nous pouvons implémenter les deux traits dans l'un ou l'autre ordre, mais c'est une erreur à implémenter pour un type sans implémenter également . Ici, nous disons que c'est un *sous-trait* de , et c'est le *supertrait de* . `Creature` `Visible` `Creature` `Visible` `Visible` `Creature`

Les sous-traits ressemblent à des sous-interfaces en Java ou en C#, en ce sens que les utilisateurs peuvent supposer que toute valeur qui implémente un sous-trait implémente également son supertrait. Mais dans Rust, un subtrait n'hérite pas des éléments associés de son supertrait; chaque trait doit toujours être dans la portée si vous voulez appeler ses méthodes.

En fait, les sous-traits de Rust ne sont en réalité qu'un raccourci pour une liaison sur . Une définition de ce genre est exactement équivalente à celle montrée précédemment: `Self Creature`

```
trait Creature where Self: Visible {
    ...
}
```

Fonctions associées au type

Dans la plupart des langages orientés objet, les interfaces ne peuvent pas inclure de méthodes statiques ou de constructeurs, mais les traits peuvent inclure des fonctions associées au type, les méthodes analogiques à statiques de Rust :

```
trait StringSet {
    /// Return a new empty set.
    fn new() -> Self;

    /// Return a set that contains all the strings in `strings`.
    fn from_slice(strings: &[&str]) -> Self;

    /// Find out if this set contains a particular `value`.
    fn contains(&self, string: &str) -> bool;

    /// Add a string to this set.
    fn add(&mut self, string: &str);
}
```

Chaque type qui implémente le trait doit implémenter ces quatre fonctions associées. Les deux premiers, et , ne prenez pas d'argument. Ils servent de constructeurs. Dans le code non générique, ces fonctions peuvent être appelées à l'aide de la syntaxe, comme toute autre fonction associée à un type : `StringSet new() from_slice() self ::`

```
// Create sets of two hypothetical types that impl StringSet:
let set1 = SortedStringSet::new();
let set2 = HashedStringSet::new();
```

Dans le code générique, c'est la même chose, sauf que le type est souvent une variable de type, comme dans l'appel à montré ici : `S::new()`

```
/// Return the set of words in `document` that aren't in `wordlist`.
fn unknown_words<S: StringSet>(document: &[String], wordlist: &S) -> S {
```

```
        let mut unknowns = S::new();
        for word in document {
            if !wordlist.contains(word) {
                unknowns.add(word);
            }
        }
        unknowns
    }
```

Comme les interfaces Java et C#, les objets traits ne prennent pas en charge les fonctions associées au type. Si vous souhaitez utiliser des objets de trait, vous devez modifier le trait, en ajoutant la liaison à chaque fonction associée qui ne prend pas un argument par référence : `&dyn StringSet where Self: Sized self`

```
    trait StringSet {
        fn new() -> Self
            where Self: Sized;

        fn from_slice(strings: &[&str]) -> Self
            where Self: Sized;

        fn contains(&self, string: &str) -> bool;

        fn add(&mut self, string: &str);
    }
```

Cette liaison indique à Rust que les objets traits sont dispensés de prendre en charge cette fonction associée particulière. Avec ces ajouts, les objets traits sont autorisés; ils ne prennent toujours pas en charge ou , mais vous pouvez les créer et les utiliser pour appeler et . La même astuce fonctionne pour toute autre méthode incompatible avec les objets traits. (Nous renoncerons à l'explication technique plutôt fastidieuse de la raison pour laquelle cela fonctionne, mais le trait est couvert au chapitre 13.) `StringSet new from_slice .contains() .add() Sized`

Appels de méthode complets

Toutes les méthodes d'appel que nous avons vues jusqu'à présent reposent sur Rust pour combler certaines pièces manquantes pour vous. Par exemple, supposons que vous écriviez ce qui suit :

```
    "hello".to_string()
```

Il est entendu que cela fait référence à la méthode du trait, dont nous appelons l'implémentation du type. Il y a donc quatre acteurs dans ce jeu : le trait, la méthode de ce trait, la mise en œuvre de cette méthode et la valeur à laquelle cette implémentation est appliquée. C'est formidable que nous n'ayons pas à épeler tout cela chaque fois que nous voulons appeler une méthode. Mais dans certains cas, vous avez besoin d'un moyen de dire exactement ce que vous voulez dire. Les appels de méthode entièrement qualifiés correspondent à la facture. `to_string to_string ToString str`

Tout d'abord, il est utile de savoir qu'une méthode n'est qu'un type spécial de fonction. Ces deux appels sont équivalents :

```
"hello".to_string()

str::to_string("hello")
```

Le deuxième formulaire ressemble exactement à un appel de fonction associé. Cela fonctionne même si la méthode prend un argument. Passez simplement comme premier argument de la fonction. `to_string self self`

Puisqu'il s'agit d'une méthode du trait standard, il existe deux autres formes que vous pouvez utiliser: `to_string ToString`

```
ToString::to_string("hello")

<str as ToString>::to_string("hello")
```

Ces quatre appels de méthode font exactement la même chose. Le plus souvent, vous n'écrirez que . Les autres formulaires sont des appels *de méthode qualifiés*. Ils spécifient le type ou le trait auquel une méthode est associée. Le dernier formulaire, avec les crochets d'angle, spécifie les deux : un *appel de méthode* complet. `value.method()`

Lorsque vous écrivez , à l'aide de l'opérateur, vous ne dites pas exactement quelle méthode vous appelez. Rust a un algorithme de recherche de méthode qui comprend cela, en fonction des types, des coercitions de référence, etc. Avec des appels complets, vous pouvez dire exactement quelle méthode vous voulez dire, et cela peut aider dans quelques cas étranges: `"hello".to_string() . to_string`

- Lorsque deux méthodes portent le même nom. L'exemple classique de hokey est celui avec deux méthodes de deux traits différents, l'une pour

le dessiner à l'écran et l'autre pour interagir avec la loi: `Outlaw .draw()`

```
outlaw.draw();  // error: draw on screen or draw pistol?

Visible::draw(&outlaw);  // ok: draw on screen
HasPistol::draw(&outlaw);  // ok: corral
```

Habituellement, vous feriez mieux de renommer l'une des méthodes, mais parfois vous ne pouvez pas.

- Lorsque le type de l'argument ne peut pas être déduit : `self`

```
let zero = 0;  // type unspecified; could be `i8`, `u8`, ...

zero.abs();  // error: can't call method `abs`
             // on ambiguous numeric type

i64::abs(zero);  // ok
```

- Lorsque vous utilisez la fonction elle-même comme valeur de fonction :

```
let words: Vec<String> =
    line.split_whitespace()  // iterator produces &str values
        .map(ToString::to_string)  // ok
        .collect();
```

- Lors de l'appel de méthodes de trait dans des macros. Nous expliquerons au chapitre 21.

La syntaxe complète fonctionne également pour les fonctions associées. Dans la section précédente, nous avons écrit pour créer un nouvel ensemble dans une fonction générique. Nous aurions aussi pu écrire ou `.S::new() StringSet::new() <S as StringSet>::new()`

Caractéristiques qui définissent les relations entre les types

Jusqu'à présent, chaque trait que nous avons examiné est autonome: un trait est un ensemble de méthodes que les types peuvent mettre en œuvre. Les traits peuvent également être utilisés dans des situations où plusieurs types doivent travailler ensemble. Ils peuvent décrire les relations entre les types.

- Le trait relie chaque type d'itérateur au type de valeur qu'il produit. `std::iter::Iterator`
- Le trait concerne les types qui peuvent être multipliés. Dans l'expression `a * b`, les valeurs et peuvent être soit du même type, soit de types différents. `std::ops::Mul a b`
- La caisse comprend à la fois un trait pour les générateurs de nombres aléatoires () et un trait pour les types qui peuvent être générés aléatoirement (). Les traits eux-mêmes définissent exactement comment ces types fonctionnent ensemble. `rand rand::Rng rand::Distribution`

Vous n'aurez pas besoin de créer des traits comme ceux-ci tous les jours, mais vous les rencontrerez dans toute la bibliothèque standard et dans des caisses tierces. Dans cette section, nous montrerons comment chacun de ces exemples est implémenté, en reprenant les fonctionnalités pertinentes du langage Rust au fur et à mesure que nous en avons besoin. La compétence clé ici est la capacité de lire les traits et les signatures de méthode et de comprendre ce qu'ils disent sur les types impliqués.

Types associés (ou fonctionnement des itérateurs)

Nous allons commencer par les itérateurs. À l'heure actuelle, chaque langage orienté objet dispose d'une sorte de support intégré pour les itérateurs, des objets qui représentent la traversée d'une séquence de valeurs.

La rouille a un trait standard, défini comme ceci: `Iterator`

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

La première caractéristique de ce trait, , est un *type associé*. Chaque type qui implémente doit spécifier le type d'élément qu'il produit. `type Item; Iterator`

La deuxième fonctionnalité, la méthode, utilise le type associé dans sa valeur de retour. renvoie un : , soit la valeur suivante de la séquence, soit lorsqu'il n'y a plus de valeurs à visiter. Le type est écrit comme , pas seulement simple , car il s'agit d'une caractéristique de chaque type d'itérateur, et non d'un type autonome. Comme toujours, et le type apparaît explicitement dans le code partout où leurs champs, méthodes, etc. sont

utilisés. next() next() Option<Self::Item> Some(item) None Self ::Item Item Item self Self

Voici à quoi ressemble l'implémentation d'un type : Iterator

```
// (code from the std::env standard library module)
impl Iterator for Args {
    type Item = String;

    fn next(&mut self) -> Option<String> {
        ...
    }
    ...
}
```

std::env::Args est le type d'itérateur renvoyé par la fonction de bibliothèque standard que nous avons utilisée au chapitre 2 pour accéder aux arguments de ligne de commande. Il produit des valeurs, donc le déclare . std::env::args() String impl type Item = String;

Le code générique peut utiliser les types associés :

```
/// Loop over an iterator, storing the values in a new vector.
fn collect_into_vector<I: Iterator>(iter: I) -> Vec<I::Item> {
    let mut results = Vec::new();
    for value in iter {
        results.push(value);
    }
    results
}
```

À l'intérieur du corps de cette fonction, Rust déduit le type de pour nous, ce qui est agréable; mais nous devons énoncer le type de retour de , et le type associé est le seul moyen de le faire. (serait tout simplement faux : on prétendrait renvoyer un vecteur d'itérateurs
!) value collect_into_vector Item Vec<I>

L'exemple précédent n'est pas du code que vous écririez vous-même, car après avoir lu le chapitre 15, vous saurez que les itérateurs ont déjà une méthode standard qui fait ceci: . Regardons donc un autre exemple avant de passer à autre chose: iter.collect()

```
/// Print out all the values produced by an iterator
fn dump<I>(iter: I)
    where I: Iterator
```

```
{
    for (index, value) in iter.enumerate() {
        println!("{}: {:?}", index, value);   // error
    }
}
```

Cela fonctionne presque. Il n'y a qu'un seul problème : il se peut qu'il ne s'agisse pas d'un type imprimable. `value`

```
error: `<I as Iterator>::Item` doesn't implement `Debug`
  |
8 |         println!("{}: {:?}", index, value);   // error
  |                                     ^^^^^
  |                           `<I as Iterator>::Item` cannot be formatted
  |                           using `{:?}` because it doesn't implement `
  |
  = help: the trait `Debug` is not implemented for `<I as Iterator>::Ite
  = note: required by `std::fmt::Debug::fmt`
help: consider further restricting the associated type
  |
5 |     where I: Iterator, <I as Iterator>::Item: Debug
  |                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Le message d'erreur est légèrement obscurci par l'utilisation par Rust de la syntaxe , qui est une façon explicite mais verbeuse de dire . Il s'agit d'une syntaxe Rust valide, mais vous aurez rarement besoin d'écrire un type de cette façon. `<I as Iterator>::Item I::Item`

L'essentiel du message d'erreur est que pour compiler cette fonction générique, nous devons nous assurer qu'elle implémente le trait, le trait de mise en forme des valeurs avec . Comme le suggère le message d'erreur, nous pouvons le faire en plaçant une limite sur
`: I::Item Debug {:?} I::Item`

```
use std::fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I::Item: Debug
{
    ...
}
```

Ou, nous pourrions écrire, « doit être un itérateur sur les valeurs »
`: I String`

```
fn dump<I>(iter: I)
    where I: Iterator<Item=String>
{
    ...
}
```

`Iterator<Item=String>` est lui-même un trait. Si vous considérez comme l'ensemble de tous les types d'itérateurs, alors est un sous-ensemble de : l'ensemble des types d'itérateurs qui produisent s. Cette syntaxe peut être utilisée partout où le nom d'un trait peut être utilisé, y compris les types d'objet de trait
: `Iterator Iterator<Item=String> Iterator String`

```
fn dump(iter: &mut dyn Iterator<Item=String>) {
    for (index, s) in iter.enumerate() {
        println!("{}: {:?}", index, s);
    }
}
```

Les traits avec des types associés, comme , sont compatibles avec les méthodes de trait, mais seulement si tous les types associés sont épelés, comme indiqué ici. Sinon, le type de pourrait être n'importe quoi, et encore une fois, Rust n'aurait aucun moyen de taper-vérifier ce code. `Iterator s`

Nous avons montré beaucoup d'exemples impliquant des itérateurs. Il est difficile de ne pas le faire; ils sont de loin l'utilisation la plus importante des types associés. Mais les types associés sont généralement utiles chaque fois qu'un trait doit couvrir plus que de simples méthodes:

- Dans une bibliothèque de pool de threads, un trait, représentant une unité de travail, peut avoir un type associé. `Task Output`
- Un trait, représentant une façon de rechercher une chaîne, peut avoir un type associé, représentant toutes les informations recueillies en faisant correspondre le motif à la chaîne : `Pattern Match`

```
trait Pattern {
    type Match;

    fn search(&self, string: &str) -> Option<Self::Match>;
}

/// You can search a string for a particular character.
impl Pattern for char {
```

```
        /// A "match" is just the location where the
        /// character was found.
        type Match = usize;

        fn search(&self, string: &str) -> Option<usize> {
            ...
        }
    }
```

Si vous êtes familier avec les expressions régulières, il est facile de voir comment aurait un type plus élaboré, probablement une structure qui inclurait le début et la longueur de la correspondance, les emplacements où les groupes entre parenthèses correspondaient, etc. `impl Pattern for RegExp Match`

- Une bibliothèque permettant d'utiliser des bases de données relationnelles peut avoir un trait avec des types associés représentant des transactions, des curseurs, des instructions préparées, etc. `Database Connection`

Les types associés sont parfaits pour les cas où chaque implémentation a *un* type spécifique connexe: chaque type de produit un type particulier de ; chaque type de recherche un type particulier de . Cependant, comme nous le verrons, certaines relations entre les types ne sont pas comme ça. `Task Output Pattern Match`

Traits génériques (ou fonctionnement de la surcharge de l'opérateur)

La multiplication dans Rust utilise ce trait:

```
/// std::ops::Mul, the trait for types that support `*`.
pub trait Mul<RHS> {
    /// The resulting type after applying the `*` operator
    type Output;

    /// The method for the `*` operator
    fn mul(self, rhs: RHS) -> Self::Output;
}
```

`Mul` est un trait générique. Le paramètre type, , est l'abréviation *de right-hand* side. `RHS`

Le paramètre type ici signifie la même chose qu'il signifie sur une structure ou une fonction: est un trait générique, et ses instances , , , etc., sont

toutes des traits différents, tout comme et sont des fonctions différentes et et sont des types différents. `Mul Mul<f64> Mul<String> Mul<Size> min:: <i32> min::<String> Vec<i32> Vec<String>`

Un seul type, par exemple, peut implémenter les deux et , et bien d'autres. Vous seriez alors en mesure de multiplier a par de nombreux autres types. Chaque implémentation aurait son propre type associé. `WindowSize Mul<f64> Mul<i32> WindowSize Output`

Les traits génériques bénéficient d'une dispense spéciale en ce qui concerne la règle orpheline: vous pouvez implémenter un trait étranger pour un type étranger, à condition que l'un des paramètres de type du trait soit un type défini dans la caisse actuelle. Donc, si vous vous êtes défini, vous pouvez implémenter pour , même si vous n'avez défini ni l'un ni l'autre ou . Ces implémentations peuvent même être génériques, telles que . Cela fonctionne parce qu'il n'y a aucun moyen qu'une autre caisse puisse définir quoi que ce soit, et donc aucun conflit entre les implémentations pourrait survenir. (Nous avons introduit la règle orpheline dans « Traits et types d'autres personnes ».) C'est ainsi que les caisses définissent les opérations arithmétiques sur les vecteurs. `WindowSize Mul<WindowSize> f64 Mul f64 impl<T> Mul<WindowSize> for Vec<T> Mul<WindowSize> nalgebra`

Le trait montré précédemment manque un détail mineur. Le vrai trait ressemble à ceci: `Mul`

```
pub trait Mul<RHS=Self> {
    ...
}
```

La syntaxe signifie que la valeur par défaut est . Si j'écris , sans spécifier le paramètre de type de ', cela signifie . Dans une limite, si j'écris , cela signifie . `RHS=Self RHS Self impl Mul for Complex Mul impl Mul<Complex> for Complex where T: Mul where T: Mul<T>`

Dans Rust, l'expression est un raccourci pour . Donc, surcharger l'opérateur dans Rust est aussi simple que de mettre en œuvre le trait. Nous montrerons des exemples dans le chapitre suivant. `lhs * rhs Mul::mul(lhs, rhs) * Mul`

Trait impl

Comme vous pouvez l'imaginer, les combinaisons de nombreux types génériques peuvent devenir désordonnées. Par exemple, la combinaison de quelques itérateurs à l'aide de combinateurs de bibliothèque standard transforme rapidement votre type de retour en une horreur :

```
use std::iter;
use std::vec::IntoIter;
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) ->
    iter::Cycle<iter::Chain<IntoIter<u8>, IntoIter<u8>>> {
        v.into_iter().chain(u.into_iter()).cycle()
}
```

Nous pourrions facilement remplacer ce type de retour poilu par un objet trait :

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> Box<dyn Iterator<Item=u8>> {
    Box::new(v.into_iter().chain(u.into_iter()).cycle())
}
```

Cependant, prendre la surcharge de l'expédition dynamique et une allocation de tas inévitable chaque fois que cette fonction est appelée juste pour éviter une signature de type laide ne semble pas être un bon échange, dans la plupart des cas.

Rust a une fonctionnalité appelée conçue précisément pour cette situation. nous permet d'« effacer » le type d'une valeur de retour, en spécifiant uniquement le ou les traits qu'elle implémente, sans envoi dynamique ni allocation de tas : `impl Trait impl Trait`

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> impl Iterator<Item=u8> {
    v.into_iter().chain(u.into_iter()).cycle()
}
```

Maintenant, plutôt que de spécifier un type particulier imbriqué de structs de combinateur d'itérateur, la signature de 's indique simplement qu'elle renvoie une sorte d'itérateur sur . Le type de retour exprime l'intention de la fonction, plutôt que ses détails d'implémentation. `cyclical_zip u8`

Cela a définitivement nettoyé le code et l'a rendu plus lisible, mais c'est plus qu'un simple raccourci pratique. L'utilisation signifie que vous pouvez modifier le type réel renvoyé à l'avenir tant qu'il implémente toujours , et tout code appelant la fonction continuera à compiler sans problème. Cela offre beaucoup de flexibilité aux auteurs de bibliothèques, car

seules les fonctionnalités pertinentes sont codées dans la signature de type. `impl Trait impl Trait Iterator<Item=u8>`

Par exemple, si la première version d'une bibliothèque utilise des combinateurs itérateurs comme dans le précédent, mais qu'un meilleur algorithme pour le même processus est découvert, l'auteur de la bibliothèque peut utiliser différents combinateurs ou même créer un type personnalisé qui implémente , et les utilisateurs de la bibliothèque peuvent obtenir les améliorations de performances sans modifier du tout leur code. `Iterator`

Il peut être tentant d'utiliser pour approximer une version distribuée statiquement du modèle d'usine couramment utilisé dans les langages orientés objet. Par exemple, vous pouvez définir un trait comme celui-ci : `impl Trait`

```
trait Shape {
    fn new() -> Self;
    fn area(&self) -> f64;
}
```

Après l'avoir implémenté pour quelques types, vous pouvez utiliser différents s en fonction d'une valeur d'exécution, comme une chaîne qu'un utilisateur entre. Cela ne fonctionne pas avec comme type de retour : `Shape impl Shape`

```
fn make_shape(shape: &str) -> impl Shape {
    match shape {
        "circle" => Circle::new(),
        "triangle" => Triangle::new(), // error: incompatible types
        "shape" => Rectangle::new(),
    }
}
```

Du point de vue de l'appelant, une fonction comme celle-ci n'a pas beaucoup de sens. est une forme d'envoi statique, de sorte que le compilateur doit connaître le type renvoyé par la fonction au moment de la compilation afin d'allouer la bonne quantité d'espace sur la pile et d'accéder correctement aux champs et aux méthodes sur ce type. Ici, il pourrait s'agir de , , ou , qui pourraient tous occuper différentes quantités d'espace et tous avoir des implémentations différentes de . `impl Trait Circle Triangle Rectangle area()`

Il est important de noter que Rust n'autorise pas les méthodes de trait à utiliser des valeurs de retour. Pour ce faire, il faudra apporter quelques améliorations au système de type des langues. Jusqu'à ce que ce travail soit terminé, seules les fonctions libres et les fonctions associées à des types spécifiques peuvent utiliser des retours. `impl Trait impl Trait`

`impl Trait` peut également être utilisé dans des fonctions qui prennent des arguments génériques. Par exemple, considérez cette fonction générique simple:

```
fn print<T: Display>(val: T) {
    println!("{}", val);
}
```

Elle est identique à cette version en utilisant : `impl Trait`

```
fn print(val: impl Display) {
    println!("{}", val);
}
```

Il y a une exception importante. L'utilisation de génériques permet aux appelants de la fonction de spécifier le type des arguments génériques, comme , alors que l'utilisation ne le fait pas. `print::<i32>(42) impl Trait`

Chaque argument se voit attribuer son propre paramètre de type anonyme, de sorte que pour les arguments est limité aux seules fonctions génériques les plus simples, sans relations entre les types d'arguments. `impl Trait impl Trait`

Consts associés

Comme les structs et les enums, les traits peuvent avoir des constantes associées. Vous pouvez déclarer un trait avec une constante associée en utilisant la même syntaxe que pour une struct ou un enum :

```
trait Greet {
    const GREETING: &'static str = "Hello";
    fn greet(&self) -> String;
}
```

Les consts associés dans les traits ont un pouvoir spécial, cependant. Comme les types et fonctions associés, vous pouvez les déclarer mais ne pas leur donner de valeur :

```
trait Float {
    const ZERO: Self;
    const ONE: Self;
}
```

Ensuite, les implémenteurs du trait peuvent définir ces valeurs :

```
impl Float for f32 {
    const ZERO: f32 = 0.0;
    const ONE: f32 = 1.0;
}

impl Float for f64 {
    const ZERO: f64 = 0.0;
    const ONE: f64 = 1.0;
}
```

Cela vous permet d'écrire du code générique qui utilise les valeurs suivantes :

```
fn add_one<T: Float + Add<Output=T>>(value: T) -> T {
    value + T::ONE
}
```

Notez que les constantes associées ne peuvent pas être utilisées avec des objets trait, car le compilateur s'appuie sur des informations de type sur l'implémentation afin de choisir la bonne valeur au moment de la compilation.

Même un trait simple sans comportement du tout, comme , peut donner suffisamment d'informations sur un type, en combinaison avec quelques opérateurs, pour implémenter des fonctions mathématiques communes comme Fibonacci: `Float`

```
fn fib<T: Float + Add<Output=T>>(n: usize) -> T {
    match n {
        0 => T::ZERO,
        1 => T::ONE,
        n => fib::<T>(n - 1) + fib::<T>(n - 2)
    }
}
```

Dans les deux dernières sections, nous avons montré différentes façons dont les traits peuvent décrire les relations entre les types. Tous ces élé-

ments peuvent également être considérés comme des moyens d'éviter les frais généraux et les downcasts de méthodes virtuelles, car ils permettent à Rust de connaître des types plus concrets au moment de la compilation.

Limites de rétro-ingénierie

L'écriture de code générique peut être un véritable slog lorsqu'il n'y a pas de trait unique qui fait tout ce dont vous avez besoin. Supposons que nous ayons écrit cette fonction non générale pour faire un calcul:

```
fn dot(v1: &[i64], v2: &[i64]) -> i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Maintenant, nous voulons utiliser le même code avec des valeurs à virgule flottante. Nous pourrions essayer quelque chose comme ceci:

```
fn dot<N>(v1: &[N], v2: &[N]) -> N {
    let mut total: N = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Pas de chance: Rust se plaint de l'utilisation et du type de . Nous pouvons exiger d'être un type qui soutient et utilise les traits et. Notre utilisation de doit changer, cependant, parce que c'est toujours un entier dans Rust; la valeur en virgule flottante correspondante est . Heureusement, il existe un trait standard pour les types qui ont des valeurs par défaut. Pour les types numériques, la valeur par défaut est toujours 0

```
:* 0 N + * Add Mul 0 0 0.0 Default
```

```
use std::ops::{Add, Mul};

fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
```

```
    total
}
```

C'est plus proche, mais cela ne fonctionne toujours pas tout à fait:

```
error: mismatched types
  |
5 | fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
  |        - this type parameter
...
8 |         total = total + v1[i] * v2[i];
  |                         ^^^^^^^^^^^^^ expected type parameter `N`,
  |                                       found associated type
  |
  = note: expected type parameter `N`
             found associated type `<N as Mul>::Output`
help: consider further restricting this bound
  |
5 | fn dot<N: Add + Mul + Default + Mul<Output = N>>(v1: &[N], v2: &[N])
  |                                 ^^^^^^^^^^^^^^^^^
```

Notre nouveau code suppose que la multiplication de deux valeurs de type produit une autre valeur de type . Ce n'est pas nécessairement le cas. Vous pouvez surcharger l'opérateur de multiplication pour renvoyer le type de votre choix. Nous devons en quelque sorte dire à Rust que cette fonction générique ne fonctionne qu'avec des types qui ont la saveur normale de la multiplication, où la multiplication renvoie un . La suggestion dans le message d'erreur est *presque* juste: nous pouvons le faire en remplaçant par , et la même chose pour : `N N N *` `N N Mul Mul<Output=N> Add`

```
fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[N]) -
{
    ...
}
```

À ce stade, les limites commencent à s'accumuler, ce qui rend le code difficile à lire. Déplaçons les limites dans une clause : `where`

```
fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default
{
    ...
}
```

Génial. Mais Rust se plaint toujours de cette ligne de code:

```
error: cannot move out of type `[N]`, a non-copy slice
  |
8 |         total = total + v1[i] * v2[i];
  |                         ^^^^^
  |                         |
  |                         cannot move out of here
  |                         move occurs because `v1[_]` has type `N`,
  |                         which does not implement the `Copy` trait
```

Comme nous n'avons pas besoin d'être un type copiable, Rust interprète comme une tentative de déplacer une valeur hors de la tranche, ce qui est interdit. Mais nous ne voulons pas du tout modifier la tranche; nous voulons simplement copier les valeurs pour les exploiter. Heureusement, tous les types numériques intégrés de Rust implémentent , nous pouvons donc simplement ajouter cela à nos contraintes sur N: `N v1[i] Copy`

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

Avec cela, le code se compile et s'exécute. Le code final ressemble à ceci :

```
use std::ops::{Add, Mul};

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default + Copy
{
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

#[test]
fn test_dot() {
    assert_eq!(dot(&[1, 2, 3, 4], &[1, 1, 1, 1]), 10);
    assert_eq!(dot(&[53.0, 7.0], &[1.0, 5.0]), 88.0);
}
```

Cela arrive parfois dans Rust: il y a une période de disputes intenses avec le compilateur, à la fin de laquelle le code a l'air plutôt agréable, comme s'il avait été un jeu d'enfant à écrire, et fonctionne à merveille.

Ce que nous avons fait ici, c'est procéder à l'ingénierie inverse des limites sur , en utilisant le compilateur pour guider et vérifier notre travail. La raison pour laquelle c'était un peu pénible est qu'il n'y avait pas un seul trait dans la bibliothèque standard qui incluait tous les opérateurs et méthodes que nous voulions utiliser. En l'occurrence, il existe une caisse open source populaire appelée qui définit un tel trait! Si nous l'avions su, nous aurions pu ajouter à notre *Cargo.toml* et écrire: `N Number num num`

```
use num::Num;

fn dot<N: Num + Copy>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::zero();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Tout comme dans la programmation orientée objet, la bonne interface rend tout agréable, dans la programmation générique, le bon trait rend tout agréable.

Pourtant, pourquoi se donner tant de mal ? Pourquoi les concepteurs de Rust n'ont-ils pas rendu les génériques plus semblables à des modèles C++, où les contraintes sont laissées implicites dans le code, à la « duck typing »?

L'un des avantages de l'approche de Rust est la compatibilité directe du code générique. Vous pouvez modifier l'implémentation d'une fonction ou d'une méthode générique publique, et si vous n'avez pas modifié la signature, vous n'avez cassé aucun de ses utilisateurs.

Un autre avantage des limites est que lorsque vous obtenez une erreur de compilateur, au moins le compilateur peut vous dire où se trouve le problème. Les messages d'erreur du compilateur C++ impliquant des modèles peuvent être beaucoup plus longs que ceux de Rust, pointant vers de nombreuses lignes de code différentes, car le compilateur n'a aucun moyen de dire qui est à blâmer pour un problème : le modèle, ou son appelant, qui peut également être un modèle, ou l'appelant de *ce* modèle...

Peut-être que l'avantage le plus important d'écrire explicitement les limites est simplement qu'elles sont là, dans le code et dans la documentation. Vous pouvez regarder la signature d'une fonction générique dans Rust et voir exactement quel type d'arguments elle accepte. On ne peut

pas en dire autant des modèles. Le travail de documentation complète des types d'arguments dans les bibliothèques C++ comme Boost est encore *plus* ardu que ce que nous avons vécu ici. Les développeurs Boost n'ont pas de compilateur qui vérifie leur travail.

Traits en tant que fondation

Les traits sont l'une des principales caractéristiques d'organisation de Rust, et avec raison. Il n'y a rien de mieux pour concevoir un programme ou une bibliothèque qu'une bonne interface.

Ce chapitre était un blizzard de syntaxe, de règles et d'explications. Maintenant que nous avons jeté les bases, nous pouvons commencer à parler des nombreuses façons dont les traits et les génériques sont utilisés dans le code Rust. Le fait est que nous n'avons fait que commencer à gratter la surface. Les deux chapitres suivants couvrent les traits communs fournis par la bibliothèque standard. Les chapitres à venir couvrent les fermetures, les itérateurs, les entrées/sorties et la simultanéité. Les traits et les génériques jouent un rôle central dans tous ces sujets.