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 tout un système.*
>
> —Donald Knuth

Une des grandes découvertesen programmation, c'est qu'il est possible d'écrire du code qui opère 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 `Vec` n'avaient jamais anticipés.
- Beaucoup de choses ont des `.write()` méthodes, y compris `File` s et `TcpStream` s. Votre code peut prendre un écrivain par référence, n'importe quel écrivain, et lui envoyer des données. Votre code n'a pas à se soucier de quel type d'écrivain il s'agit. Plus tard, si quelqu'un ajoute un nouveau type de rédacteur, votre code le supportera déjà.

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 en vogue des années 1970. À présent, il est effectivement universel. Rust prend en charge le polymorphisme avec deux fonctionnalités connexes : les traits et les génériques. Ces concepts seront familiers à de nombreux programmeurs, mais Rust adopte une nouvelle approche inspirée des classes de types de Haskell.

Les traits sont le point de vue de Rust sur les interfaces ou les classes de base abstraites. Au début, elles ressemblent à des interfaces en Java ou en C#. Le trait pour écrire des octets s'appelle `std::io::Write` , et sa définition dans la bibliothèque standard commence comme ceci :

```
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 propose plusieurs méthodes; nous n'avons montré que les trois premiers.

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

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

```
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 dans les traits que ce que nous avons laissé entendre jusqu'à présent. Nous les utiliserons pour ajouter des méthodes d'extension aux types existants, même aux types intégrés comme `str` et `bool`. Nous expliquerons pourquoi l'ajout d'un trait à un type ne coûte pas de mémoire supplémentaire et comment utiliser des traits sans surcoût d'appel de méthode virtuelle. Nous verrons que les traits intégrés sont le crochet dans le langage fourni par Rust pour la surcharge des opérateurs et d'autres fonctionnalités. Et nous couvrirons le `Self` type, les fonctions associées et les types associés, trois fonctionnalités que Rust a extraites de Haskell qui résolvent élégamment les problèmes que d'autres langages traitent avec des solutions de contournement et des hacks.

*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
    }
}
```

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

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

Utilisation des caractéristiques

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

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

- `std::fs::File` implémente le `Write` trait ; il écrit des octets dans un fichier local. `std::net::TcpStream` écrit sur une connexion réseau. `Vec<u8>` implémente également `Write` . Chaque `.write()` appel sur un vecteur d'octets ajoute des données à la fin.

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

Il existe une règle inhabituelle concernant les méthodes de trait : le trait lui-même doit être dans la portée. Sinon, toutes ses méthodes sont masquées :

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

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

```
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 `u32` et `str`. Les caisses tierces peuvent faire la même chose. Évidemment, cela pourrait entraîner des conflits de nommage ! Mais puisque Rust vous oblige à importer les traits que vous prévoyez d'utiliser, les caisses sont libres de profiter de cette superpuissance. Pour obtenir un conflit, vous devez importer deux traits qui ajoutent une méthode portant le même nom au même type. C'est rare en pratique. (Si vous rencontrez un conflit, vous pouvez préciser ce que vous voulez en utilisant la syntaxe de méthode entièrement qualifiée, abordée plus loin dans le chapitre.)

La raison `Clone` et `Iterator` les méthodesfonctionnent sans aucune importation spéciale, c'est qu'ils sont toujours dans la portée par défaut : ils font partie du prélude standard, des noms que Rust importe automatiquement dans chaque module. En fait, le prélude est surtout une sélection soigneusement choisie de traits. Nous en couvrirons beaucoup au chapitre 13.

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. Autrement dit, il n'y a pas de polymorphisme ici. Il est évident qu'il `buf` s'agit d'un vecteur, pas d'un fichier ou d'une connexion réseau. Le compilateur peut émettre un simple appel à `Vec<u8>::write()`. Il peut même intégrer la méthode. (C++ et C# feront souvent la même chose, bien que la possibilité de sous-classement l'empêche parfois.) Seuls les appels traversants `&mut dyn Write` entraînent la surcharge d'une répartition dynamique, également connue sous le nom d'appel de méthode virtuelle, qui est indiquée par le mot- `dyn` clé dans le type. `dyn Write` est connu comme un *objet trait* ; nous examinerons les détails techniques des objets trait, et comment ils se comparent aux fonctions génériques, dans les sections suivantes.

Objets de trait

Il y a deux façons d'utiliser les traitsécrire du code polymorphe en Rust : objets traits et génériques. Nous présenterons d'abord les objets de trait et passerons aux génériques dans la section suivante.

Rust 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 siz
```

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

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

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

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 de trait inclut un peu d'informations supplémentaires sur le type du référent. C'est strictement pour l'usage propre de Rust dans les coulisses : lorsque vous appelez `writer.write(data)` , Rust a besoin des informations de type pour appeler dynamiquement la bonne `write` méthode en fonction du type de `*writer` . Vous ne pouvez pas interroger directement les informations de type et Rust ne prend pas en charge la conversion descendante de l'objet de trait `&mut dyn Write` en un type concret tel que `Vec<u8>` .

Disposition des objets de trait

En mémoire, un objet traitest 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 occupe donc deux mots machine, comme le montre la figure 11-1 .

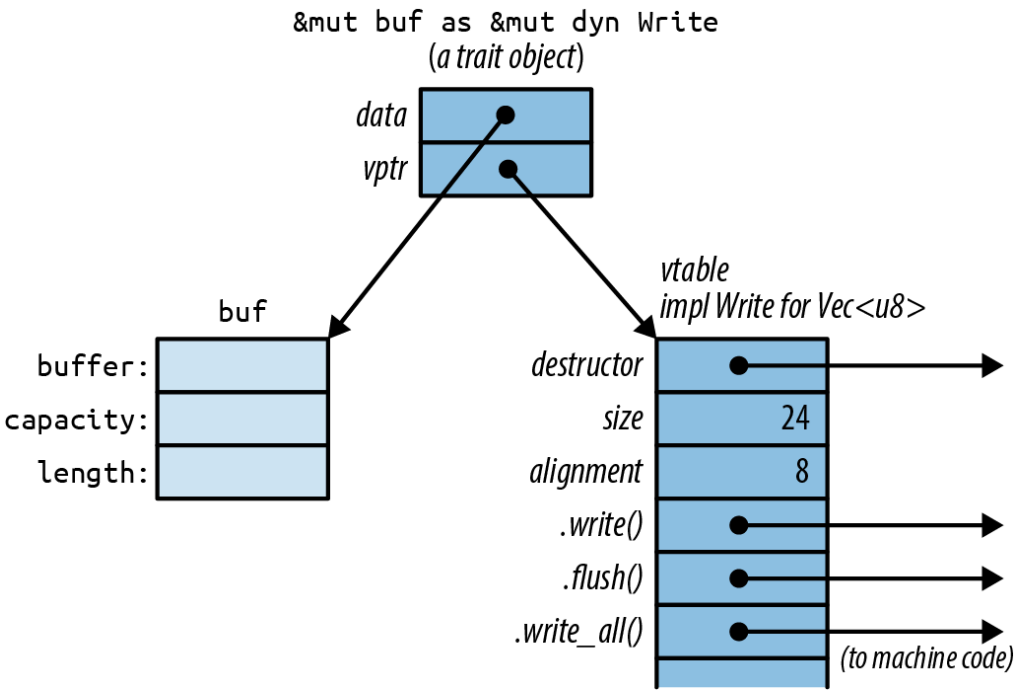

Illustration 11-1. Objets de trait en mémoire

C++ possède également ce type d'informations de type à l'exécution. C'est ce qu'on appelle une *table virtuelle*, ou *vtable* . En Rust, comme en C++, la vtable est générée une seule fois, au moment de la compilation, et partagée par tous les objets du même type. Tout ce qui apparaît dans la teinte plus foncée de la figure 11-1 , y compris la 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 la vtable lorsque vous appelez une méthode d'un objet trait, pour déterminer quelle implémentation 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 des pointeurs gras à la place. La structure elle-même ne contient que ses champs. De cette façon, une structure peut implémenter des dizaines de traits sans contenir des dizaines de vptrs. Même des types comme `i32` , qui ne sont pas assez grands pour accueillir un vptr, peuvent implémenter des traits.

Rust convertit automatiquement les références ordinaires en objets trait si nécessaire. C'est pourquoi nous pouvons passer `&mut local_file` à `say_hello` dans cet exemple :

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

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

De même, Rust se fera un plaisir de convertir a `Box<File>` en a `Box<dyn Write>` , une valeur qui possède un écrivain dans le tas :

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

`Box<dyn Write>` , comme `&mut dyn Write` , est un gros pointeur : il contient l'adresse du rédacteur lui-même et l'adresse de la vtable. Il en va de même pour les autres types de pointeurs, comme `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, `File`), il ajoute donc simplement l'adresse de la vtable appropriée, transformant le pointeur normal en un pointeur gras.

Fonctions génériques et paramètres de type

Au débutde ce chapitre, nous avons montré une `say_hello()` fonctionqui a pris un objet trait comme argument. Réécrivons cette fonction comme une fonction générique :

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

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

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

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

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 examinant les arguments. Ici, la `say_hello` fonction

générique attend un `&mut W` argument, et nous lui transmettons un `&mut File`, donc Rust en déduit que `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 assez bon. Comment envisageons-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 prendre en charge les opérations `Hash` et `Eq`. Les bornes sur `T` doivent inclure celles-ci ainsi que `Debug`. La syntaxe pour cela utilise le `+` signe :

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

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

Certains types implement `Debug`, d'autres implement `Hash`, d'autres support `Eq`, et quelques-uns, comme `u32` et `String`, implémentent les trois, comme le montre la figure 11-2 .

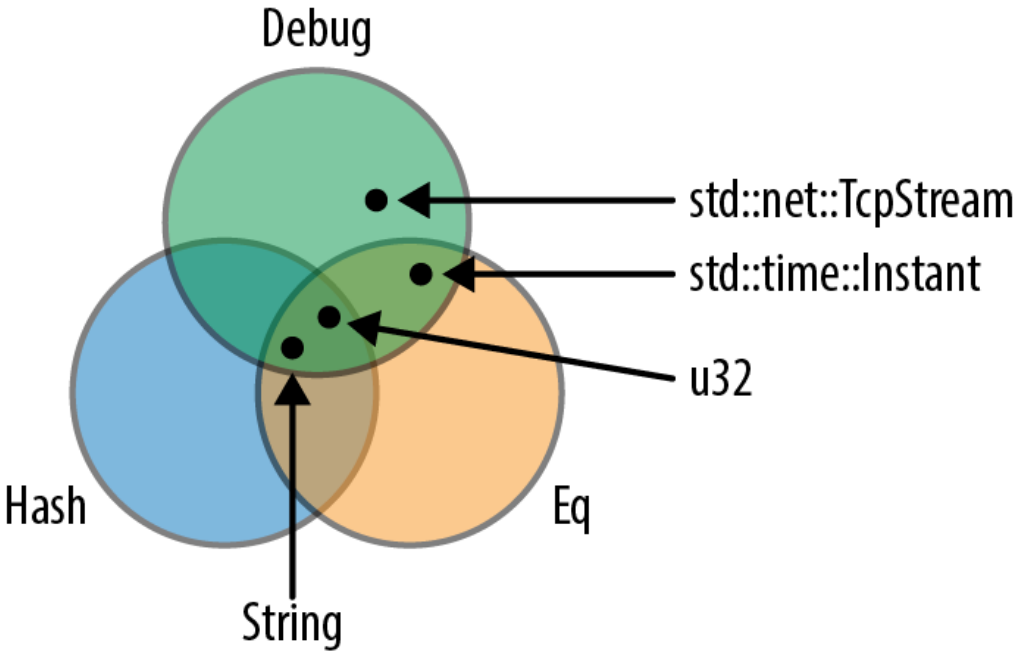

Illustration 11-2. Les traits comme ensembles de types

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 ça.

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 devenir si longues qu'elles sont pénibles 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 `M` et `R` sont toujours déclarés au début, mais les limites sont déplacées sur des lignes séparées. Ce type de `where` clause est également autorisé sur les structures génériques, les énumérations, les alias de type et les méthodes, partout où les limites sont autorisées.

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

aussi intensive.

« Recevoir des références en tant qu'arguments de fonction » a introduit la syntaxe pour la durée de vieparamètres. 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, `target` et `candidates`. Les deux sont des références, et nous leur donnons des durées de vie distinctes `'t` et `'c` (comme indiqué dans "Paramètres de durée de vie distincts"). De plus, la fonction fonctionne avec n'importe quel type `P` qui implémente le `MeasureDistance` trait, nous pouvons donc l'utiliser sur des `Point2d` valeurs dans un programme et `Point3d` des valeurs dans un autre.

Les durées de vie n'ont jamais d'impact sur le code machine. Deux appels `nearest()` utilisant le même type `P`, 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.

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

```
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 `<const N: usize>` indique que la fonction `dot_product` attend un paramètre générique `N`, qui doit être un `usize`. Étant donné `N`, la fonction prend deux arguments de type `[f64; N]`, et additionne les produits de leurs éléments correspondants. Ce qui

se distingue d'un argument `N` ordinaire , c'est que vous pouvez l'utiliser dans les types dans la signature ou le corps de . `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 entendu, 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 aborderons les traits génériques plus loin dans ce chapitre.

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

Lequel utiliser

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

Les objets de trait sont le bon choix chaque fois que vous avez besoin d'une collection de valeurs de types mixtes, toutes ensemble. Il est techniquement possible de faire de la salade générique :

```
trait Vegetable {
    ...
}

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

Cependant, il s'agit d'une conception assez 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 choses. Un de vos auteurs a déjà payé 14 $ pour un `Salad<IcebergLettuce>` et ne s'est jamais vraiment remis de l'expérience.

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

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

Les objets trait sont la solution :

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

Chacun `Box<dyn Vegetable>` 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 malheureuse métaphore mixte d'avoir des boîtes dans sa nourriture, c'est précisément ce qu'il faut, 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. sur.

Une autre raison possible d'utiliser des objets trait est de réduire la quantité totale de code compilé. Rust peut devoir compiler une fonction générique plusieurs fois, une fois pour chaque type avec lequel elle est utili-

sée. Cela pourrait rendre le binaire volumineux, un phénomène appelé *gonflement du code* dans les cercles C++ . De nos jours, la mémoire est abondante et la plupart d'entre nous ont le luxe d'ignorer la taille du code ; mais des environnements contraints existent.

En dehors des situations impliquant des salades ou des environnements à faibles ressources, les génériques ont trois avantages importants par rapport aux objets de trait, avec pour résultat que dans Rust, les génériques sont le choix le plus courant.

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

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

Ou considérez cet appel de fonction générique :

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

`std::io::sink()` renvoie un écrivain de type `Sink` qui supprime silencieusement tous les octets qui lui sont écrits.

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

Ou, Rust pourrait regarder 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 entièrement cet appel de fonction.

Comparez cela au comportement avec les objets trait. Rust ne sait jamais sur quel type de valeur un objet de trait pointe jusqu'au moment de l'exécution. Ainsi, même si vous transmettez a `Sink` , la surcharge liée à l'appel de méthodes virtuelles et à la recherche d'erreurs s'applique toujours.

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

Le troisième avantage des génériques est qu'il est facile de lier un paramètre de type générique à plusieurs traits à la fois, comme notre `top_ten` fonction l'a fait lorsqu'elle a demandé à son `T` paramètre d'implémenter `Debug + Hash + Eq` . Les objets de trait ne peuvent pas faire cela : les types comme `&mut (dyn Debug + Hash + Eq)` 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 complexe.)

Définir et mettre en œuvre les traits

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

```
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 `impl` blocs de caractéristiques :

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

Méthodes par défaut

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

```rust
/// 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 `Write` trait pour `Sink` :

```rust
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 `Visible` trait. Mais nous avons aussi vu que le `Write` trait a une `write_all` méthode :

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

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

```rust
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 `write` et `flush` sont les méthodes de base que chaque écrivain doit mettre en œuvre. Un écrivain peut également implémenter `write_all`, mais si ce n'est pas le cas, l'implémentation par défaut présentée précédemment sera utilisée.

Vos propres traits 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 `Iterator` trait, qui a une méthode requise (`.next()`) et des dizaines de méthodes par défaut. Le chapitre 15 explique pourquoi.

Traits et autres types de personnes

Rouillervous 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 ajouterune 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 `is_emoji` méthode n'est visible que lorsqu'elle `IsEmoji` est dans la portée.

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

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

```
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 `impl<W: Write> WriteHtml for W` signifie "pour chaque type `W` qui implémente `Write`, voici une implémentation de `WriteHtml` for `W`."

La `serde` bibliothèqueoffre un bel exemple de l'utilité d'implémenter des traits définis par l'utilisateur sur des types standard. `serde` 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, `Serialize`, qui est implémenté pour chaque type de données pris en charge par la bibliothèque. Ainsi, dans le `serde` code source, il existe un code implémentant `Serialize` pour les types `bool`, `i8`, `i16`, `i32`, array et tuple, etc., à travers toutes les structures de données standard telles que `Vec` et `HashMap`.

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

```
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 précédemment 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 des orphelins*. Cela aide Rust à s'assurer que les implémentations de traits sont uniques. Votre code ne peut pas `impl Write for u8`, car les deux `Write` et `u8` sont définis dans la bibliothèque standard. Si Rust laissait les caisses faire cela, il pourrait y avoir plusieurs implémentations de `Write` for `u8`, dans différentes caisses, et Rust n'aurait aucun moyen raisonnable de décider quelle implémentation utiliser pour un appel de méthode donné..

(C++ a une restriction d'unicité similaire : la règle de définition unique. En mode 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 traitpeut utiliser le mot-clé `Self` comme type. Le `Clone` trait standard, par exemple, ressemble à ceci (légèrement simplifié):

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

Utiliser `Self` comme type de retour ici signifie que le type de `x.clone()` est le même que le type de `x`, quel qu'il soit. Si `x` est un `String`, alors le type de `x.clone()` est `String` —not `dyn Clone` ou tout autre type clonable.

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 `impl` , `Self` est simplement un alias pour `CherryTree` , et dans le second, c'est un alias pour `Mammoth` . Cela signifie que nous pouvons assembler deux cerisiers ou deux mammouths, pas que nous pouvons créer un hybride mammouth-cerisier. Le type de `self` et le type de `other` doivent correspondre.

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

```
// 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 à mesure que nous approfondirons les fonctionnalités avancées des traits. Rust rejette ce code car il n'a aucun moyen de vérifier le type de l'appel `left.splice(right)` . L'intérêt des objets trait est que le type n'est connu qu'au moment de l'exécution. Rust n'a aucun moyen de savoir au moment de la compilation si `left` et `right` seront du même type, comme requis.

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

Maintenant, si nous avions voulu un épissage génétiquement improbable, nous aurions pu concevoir un trait respectueux de l'objet :

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

Ce trait est compatible avec les objets trait. Il n'y a aucun problème pour vérifier le type des appels à cette `.splice()` méthodecar le type de l'argument `other` n'est pas obligé de correspondre au type de `self`, tant que les deux typessont `MegaSpliceable`.

Sous-traits

Nous pouvons déclarerqu'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;
    ...
}
```

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

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

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

Nous pouvons implémenter les deux traits dans n'importe quel ordre, mais c'est une erreur d'implémenter `Creature` pour un type sans également implémenter `Visible`. Ici, on dit que `Creature` c'est un sous- *trait* de `Visible`, et c'est `Visible` le `Creature` sur - *trait de*.

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

En fait, les sous-traits de Rust ne sont en réalité qu'un raccourci pour un lien sur `Self`. Une définition de `Creature` like this est exactement équivalente à celle présentée précédemment :

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

Fonctions associées au type

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

```
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 `StringSet` trait doit implémenter ces quatre fonctions associées. Les deux premiers, `new()` et `from_slice()`, neprendre un `self` argument. Ils servent de constructeurs. Dans du code non générique, ces fonctions peuvent être appelées à l'aide de la `::` syntaxe, comme n'importe quelle autre fonction associée à un type :

```
// 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 `S::new()` montré ici :

```
/// 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 trait ne prennent pas en charge les fonctions associées au type. Si vous souhaitez utiliser des `&dyn StringSet` objets trait, vous devez modifier le trait, en ajoutant le `where Self: Sized` lien à chaque fonction associée qui ne prend pas d' `self` argument par référence :

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

Appels de méthode entièrement qualifiés

Tous les cheminspour appeler les méthodes de trait que nous avons vues jusqu'à présent, s'appuyer sur Rust pour remplir certaines pièces manquantes pour vous. Par exemple, supposons que vous écriviez ce qui suit :

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

Il est entendu que `to_string` fait référence à la `to_string` méthode du `ToString` trait, dont nous appelons l' `str` implémentation du type. Il y a donc quatre joueurs dans ce jeu : le trait, la méthode de ce trait, la mise en œuvre de cette méthode et la valeur à laquelle cette mise en œuvre est appliquée. C'est bien que nous n'ayons pas à épeler tout cela à 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 font l'affaire.

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

```
"hello".to_string()

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

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

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

```
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'écrivez que `value.method()` . Les autres formes sont des appels de méthode *qualifiés* . Ils spécifient le type ou le trait auquel une méthode est associée. La dernière forme, avec les chevrons, spécifie les deux : un appel de méthode *entièrement qualifié* .

Lorsque vous écrivez `"hello".to_string()` , en utilisant l' `.` opérateur, vous ne dites pas exactement quelle `to_string` méthode vous appelez. Rust a un algorithme de recherche de méthode qui calcule cela, en fonction des types, des coercitions déréférencées, etc. Avec des appels entièrement qualifiés, vous pouvez dire exactement de quelle méthode vous parlez, et cela peut aider dans quelques cas étranges :

- Lorsque deux méthodes portent le même nom. L'exemple classique du hokey est le `Outlaw` avec deux `.draw()` méthodes de deux traits dif-

férents, une pour le dessiner sur l'écran et une pour interagir avec la loi :

```
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' `self` argument ne peut pas être déduit :

```
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
```

- Lors de l'utilisation de 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 vous expliquerons au chapitre 21 .

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

Caractéristiques qui définissent les relations entre les types

Jusqu'à présent, chaque traitnous avons examiné les supports isolés : un trait est un ensemble de méthodes que les types peuvent implémenter. Les traits peuvent également être utilisés dans des situations où plusieurs types doivent fonctionner ensemble. Ils peuvent décrire les relations entre les types.

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

Vous n'aurez pas besoin de créer des traits comme ceux-ci tous les jours, mais vous les rencontrerez dans la bibliothèque standard et dans des caisses tierces. Dans cette section, nous montrerons comment chacun de ces exemples est implémenté, en sélectionnant 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 présent, chaque langage orienté objet a une sorte de support intégré pour les itérateurs, des objets qui représentent le parcours d'une séquence de valeurs.

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

```
pub trait Iterator {
    type Item;

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

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

La deuxième fonctionnalité, la `next()` méthode, utilise le type associé dans sa valeur de retour. `next()` Retouran `Option<Self::Item>` : soit `Some(item)` , la valeur suivante dans la séquence, soit `None` lorsqu'il n'y a plus de valeurs à visiter. Le type s'écrit `Self::Item` , pas simplement plain `Item` , car il `Item` s'agit d'une fonctionnalité de chaque type d'itérateur, et non d'un type autonome. Comme toujours, `self` et le `Self` type

apparaît explicitement dans le code partout où leurs champs, méthodes, etc. sont utilisés.

Voici à quoi cela ressemble à implémenter `Iterator` pour un type :

```
// (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 `std::env::args()` que nous avons utilisée au chapitre 2 pour accéder aux arguments de la ligne de commande. Il produit des `String` valeurs, donc le `impl` declares `type Item = String;` .

Génériquele code peut utiliser des 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
}
```

Dans le corps de cette fonction, Rust déduit le type de `value` pour nous, ce qui est bien ; mais nous devons préciser le type de retour de `collect_into_vector` , et le `Item` type associé est le seul moyen de le faire. (`Vec<I>` serait tout simplement faux : nous prétendrions renvoyer un vecteur d'itérateurs !)

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 cela : `iter.collect()` . Prenons donc un autre exemple avant de poursuivre :

```
/// 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 : `value` ce n'est peut-être pas un type imprimable.

```
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 `Deb
  |
  = help: the trait `Debug` is not implemented for `<I as Iterator>::Item`
  = 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 `<I as Iterator>::Item`, qui est une manière explicite mais verbeuse de dire `I::Item`. Il s'agit d'une syntaxe Rust valide, mais vous aurez rarement besoin d'écrire un type de cette façon.

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

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

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

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

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

```
{
    ...
}
```

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

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

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

Nous avons montré de nombreux exemples impliquant des itérateurs. C'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 lorsqu'un trait doit couvrir plus que de simples méthodes :

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

```
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 `impl Pattern for RegExp` aurait un `Match` type plus élaboré, probablement une structure qui inclurait le début et la longueur de la correspondance, les emplacements où les groupes entre parenthèses correspondent, etc.

- Une bibliothèque pour travailler avec des bases de données relationnelles peut avoir un `DatabaseConnection` trait avec des types associés représentant des transactions, des curseurs, des instructions préparées, etc.

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

Traits génériques (ou comment fonctionne la surcharge d'opérateur)

Multiplicationdans 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 génériquecaractéristique. Le paramètre de type, `RHS` , est l'abréviation de *righthand side* .

Le paramètre de type signifie ici la même chose que sur une structure ou une fonction : `Mul` est un trait générique, et ses instances `Mul<f64>` , `Mul<String>` , `Mul<Size>` , etc., sont tous des traits différents, tout comme `min::<i32>` et `min::<String>` sont des fonctions différentes et `Vec<i32>` et `Vec<String>` sont des types différents.

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

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

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

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

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

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

Trait de mise en œuvre

Comme vous pouvez l'imaginer, les combinaisonsde nombreux types génériques peuvent devenir désordonnés. 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 la répartition dynamique et une allocation de tas inévitable à 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 `impl Trait` conçue précisément pour cette situation. `impl Trait` permet d'"effacer" le type d'une valeur de retour, en spécifiant uniquement le ou les traits qu'elle implémente, sans dispatch dynamique ni allocation de tas :

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

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

Cela a définitivement nettoyé le code et l'a rendu plus lisible, mais `impl Trait` c'est plus qu'un simple raccourci pratique. Utiliser `impl Trait` signifie que vous pouvez modifier le type réel renvoyé à l'avenir tant qu'il implémente toujours `Iterator<Item=u8>` , et tout code appelant la fonction continuera à se compiler sans problème. Cela offre une grande flexibilité aux auteurs de bibliothèques, car seule la fonctionnalité pertinente est encodée dans la signature de type.

Par exemple, si la première version d'une bibliothèque utilise des combinateurs itérateurs comme dans le précédent, mais qu'un meilleur algo-

rithme pour le même processus est découvert, l'auteur de la bibliothèque peut utiliser différents combinateurs ou même créer un type personnalisé qui implémente `Iterator`, et les utilisateurs de la bibliothèque peuvent obtenir les améliorations de performances sans changer du tout leur code.

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

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

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

```
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. `impl Trait` est une forme de répartition 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 de ce type. Ici, il pourrait s'agir `Circle` de, `Triangle`, ou `Rectangle`, qui pourraient tous occuper des quantités d'espace différentes et tous avoir des implémentations différentes de `area()`.

Il est important de noter que Rust n'autorise pas les méthodes de trait à utiliser des `impl Trait` valeurs de retour. La prise en charge de cela nécessitera quelques améliorations dans le système de type des langues. Tant que ce travail n'est pas terminé, seules les fonctions libres et les fonctions associées à des types spécifiques peuvent utiliser des `impl Trait` retours.

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

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

Il est identique à cette version utilisant `impl Trait` :

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

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

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

Const associés

Comme les structures et les énumérations, les traits peuvent avoir des constantes associées. Vous pouvez déclarer un trait avec une constante associée en utilisant la même syntaxe que pour une structure ou une énumération :

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

Les constantes associées aux traits ont cependant un pouvoir spécial. Comme les types et fonctions associés, vous pouvez les déclarer mais 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 génériquequi utilise ces valeurs :

```
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 les informations de type concernant l'implémentation afin de choisir la bonne valeur au moment de la compilation.

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

```
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 manières dont les traits peuvent décrire les relations entre les types. Tous ces éléments peuvent également être considérés comme des moyens d'éviter les surcharges et les downcasts des méthodes virtuelles, car ils permettent à Rust de connaître des types plus concrets au moment de la compilation..

Limites de la rétro-ingénierie

Ecrire du code génériquepeut être une véritable corvée 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 générique pour effectuer des calculs :

```
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 `0`. Nous pouvons exiger `N` d'être un type qui supporte `+` et `*` utilise les traits `Add` et `Mul` Notre utilisation de `0` doit changer, cependant, car `0` est toujours un entier dans Rust ; la valeur à virgule flottante correspondante est `0.0`. Heureusement, il existe un `Default` trait standard pour les types qui ont des valeurs par défaut. Pour les types numériques, la valeur par défaut est toujours 0 :

```
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
}
```

Ceci est plus proche, mais 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 `N` produit une autre valeur de type `N`. 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 `N * N` renvoie un `N`. La suggestion dans le message d'erreur est *presque* juste : nous pouvons le faire en remplaçant `Mul` par `Mul<Output=N>`, et de même pour `Add` :

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

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

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

Super. 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
```

Puisque nous n'avons pas besoin `N` d'être un type copiable, Rust interprète `v1[i]` 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 juste copier les valeurs pour les opérer. Heureusement, tous les types numériques intégrés de Rust implémentent `Copy`, nous pouvons donc simplement ajouter cela à nos contraintes sur N :

```
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 d'intenses discussions avec le compilateur, à la fin de laquelle le code a l'air plutôt sympa, comme s'il avait été un jeu d'enfant à écrire, et s'exécute à merveille.

Ce que nous avons fait ici, c'est rétro-concevoir les limites sur `N`, 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 `Number` trait dans la bibliothèque standard qui incluait tous les opérateurs et méthodes que nous voulions utiliser. Il se trouve qu'il existe une caisse open source populaire appelée `num` qui définit un tel trait! Si nous l'avions su, nous aurions pu ajouter `num` à notre *Cargo.toml* et écrire :

```
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 tout ce mal ? Pourquoi les concepteurs de Rust n'ont-ils pas fait en sorte que les génériques ressemblent davantage 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 ascendante 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 bornes est que lorsque vous obtenez une erreur du 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 pour cause. 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 avons seulement commencé à gratter la surface. Les deux chapitres suivants couvrent les traits communs fournis par la bibliothèque standard. Les prochains chapitres traitent des fermetures, des itérateurs, des entrées/sorties et de la concurrence. Les traits et les génériques jouent un rôle central dans tous ces sujets.