Chapitre 12. Surcharge de l'opérateur

Dans le Mandelbrottraceur d'ensembles que nous avons montré au chapitre 2 , nous avons utilisé le type `num` de caisse `Complex` pour représenter un nombre sur le plan complexe :

```
#[derive(Clone, Copy, Debug)]
struct Complex<T> {
    /// Real portion of the complex number
    re:T,

    /// Imaginary portion of the complex number
    im:T,
}
```

Nous avons pu additionner et multiplier `Complex` des nombres comme n'importe quel type numérique intégré, en utilisant les opérateurs Rust `+` et : `*`

```
z = z * z + c;
```

Vous pouvez créer vos propres typesprennent également en charge les opérateurs arithmétiques et autres, simplement en implémentant quelques traits intégrés. C'est ce qu'on appelle la *surcharge d'opérateur* et l'effet ressemble beaucoup à la surcharge d'opérateur en C++, C#, Python et Ruby.

Les traitspour la surcharge des opérateurs appartiennent à quelques catégories en fonction de la partie du langage qu'ils prennent en charge, comme indiqué dans le Tableau 12-1 . Dans ce chapitre, nous couvrirons chaque catégorie. Notre objectif n'est pas seulement de vous aider à bien intégrer vos propres types dans le langage, mais aussi de vous donner une meilleure idée de la façon d'écrire des fonctions génériques comme la fonction de produit scalaire décrite dans "Reverse-Engineering Bounds" qui fonctionnent sur les types le plus naturellement utilisé via ces opérateurs. Le chapitre devrait également donner un aperçu de la façon dont certaines fonctionnalités du langage lui-même sont implémentées.

Tableau 12-1. Résumé des caractéristiques de surcharge d'opérateur

Catégorie	Caractéristique	Opérateur
Opérateurs unaires	`std::ops::Neg`	`-x`
	`std::ops::Not`	`!x`
Opérateurs arithmétiques	`std::ops::Add`	`x + y`
	`std::ops::Sub`	`x - y`
	`std::ops::Mul`	`x * y`
	`std::ops::Div`	`x / y`
	`std::ops::Rem`	`x % y`
Opérateurs au niveau du bit	`std::ops::BitAnd`	`x & y`
	`std::ops::BitOr`	`x | y`
	`std::ops::BitXor`	`x ^ y`
	`std::ops::Shl`	`x << y`
	`std::ops::Shr`	`x >> y`
Opérateurs arithmétiques d' affectation composée	`std::ops::AddAssign`	`x += y`

Catégorie	Caractéristique	Opérateur
	`std::ops::SubAssign`	`x -= y`
	`std::ops::MulAssign`	`x *= y`
	`std::ops::DivAssign`	`x /= y`
	`std::ops::RemAssign`	`x %= y`
Opérateurs bit à bit d'affectation composée	`std::ops::BitAndAssign`	`x &= y`
	`std::ops::BitOrAssign`	`x |= y`
	`std::ops::BitXorAssign`	`x ^= y`
	`std::ops::ShlAssign`	`x <<= y`
	`std::ops::ShrAssign`	`x >>= y`
Comparaison	`std::cmp::PartialEq`	`x == y`,`x != y`
	`std::cmp::PartialOrd`	`x < y`, `x <= y`, `x > y`, `x >= y`
Indexage	`std::ops::Index`	`x[y]`, `&x[y]`
	`std::ops::IndexMut`	`x[y] = z`, `&mut x[y]`

Opérateurs arithmétiques et binaires

En rouille, l'expression `a + b` est en fait un raccourci pour `a.add(b)`, un appel à la méthode du trait `add` de la bibliothèque standard. `std::ops::Add` Les types numériques standard de Rust implémentent tous `std::ops::Add`. Pour que l'expression `a + b` fonctionne pour `Complex` les valeurs, la `num` caisse implémente `Complex` également ce trait pour. Des traits similaires couvrent les autres opérateurs : `a * b` est un raccourci pour `a.mul(b)`, une méthode du `std::ops::Mul` trait, `std::ops::Neg` couvre l'opérateur de négation du préfixe `-`, etc.

Si vous voulez essayer d'écrire `z.add(c)`, vous devrez mettre le `Add` trait dans la portée afin que sa méthode soit visible. Cela fait, vous pouvez traiter toutes les opérations arithmétiques comme des appels de fonction : [1]

```
use std:: ops::Add;

assert_eq!(4.125f32.add(5.75), 9.875);
assert_eq!(10.add(20), 10 + 20);
```

Voici la définition de `std::ops::Add` :

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

En d'autres termes, le trait `Add<T>` est la capacité d'ajouter une `T` valeur à vous-même. Par exemple, si vous souhaitez pouvoir ajouter des valeurs `i32` et à votre type, votre type doit implémenter à la fois et . Le paramètre de type du trait est par défaut , donc si vous implémentez l'addition entre deux valeurs du même type, vous pouvez simplement écrire pour ce cas. Le type associé décrit le résultat de l'addition. `u32 Add<i32> Add<u32> Rhs Self Add Output`

Par exemple, pour pouvoir additionner `Complex<i32>` des valeurs, `Complex<i32>` il faut implémenter `Add<Complex<i32>>`. Puisque nous ajoutons un type à lui-même, nous écrivons simplement `Add` :

```
use std:: ops::Add;

impl Add for Complex<i32> {
    type Output = Complex<i32>;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
```

```
            im:self.im + rhs.im,
        }
    }
}
```

Bien sûr, nous ne devrions pas avoir à implémenter `Add` séparément pour `Complex<i32>`, `Complex<f32>`, `Complex<f64>`, etc. Toutes les définitions auraient exactement la même apparence, à l'exception des types impliqués, nous devrions donc être en mesure d'écrire une seule implémentation générique qui les couvre toutes, tant que le type des composants complexes eux-mêmes prend en charge l'addition :

```
use std:: ops::Add;

impl<T> Add for Complex<T>
where
    T: Add<Output = T>,
{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
            im:self.im + rhs.im,
        }
    }
}
```

En écrivant `where T: Add<Output=T>`, nous restreignons `T` aux types qui peuvent être ajoutés à eux-mêmes, donnant une autre `T` valeur. C'est une restriction raisonnable, mais nous pourrions encore assouplir les choses : le `Add` trait n'exige pas que les deux opérandes de `+` aient le même type, et il ne contraint pas non plus le type de résultat. Ainsi, une implémentation au maximum générique laisserait les opérandes gauche et droit varier indépendamment et produirait une `Complex` valeur de n'importe quel type de composant produit par l'addition :

```
use std:: ops::Add;

impl<L, R> Add<Complex<R>> for Complex<L>
where
    L: Add<R>,
{
    type Output = Complex<L:: Output>;
    fn add(self, rhs: Complex<R>) -> Self:: Output {
        Complex {
            re: self.re + rhs.re,
            im:self.im + rhs.im,
```

```
            }
        }
    }
```

En pratique, cependant, Rust a tendance à éviter de prendre en charge les opérations de type mixte. Étant donné que notre paramètre de type `L` doit implémenter `Add<R>`, il suit généralement cela `L` et `R` sera du même type : il n'y a tout simplement pas beaucoup de types disponibles pour `L` cet implémentation. Donc, en fin de compte, cette version générique maximale peut ne pas être beaucoup plus utile que la définition générique précédente, plus simple.

Les traits intégrés de Rust pour les opérateurs arithmétiques et binaires se répartissent en trois groupes : les opérateurs unaires, les opérateurs binaires et les opérateurs d'affectation composés. Au sein de chaque groupe, les traits et leurs méthodes ont tous la même forme, nous allons donc couvrir un exemple de chaque.

Opérateurs unaires

De côtéà partir de l'opérateur de déréférencement `*`, que nous aborderons séparément dans « Deref et DerefMut », Rust a deux opérateurs unaires que vous pouvez personnaliser, illustrés dans le tableau 12-2.

Tableau 12-2. Traits intégrés pour les opérateurs unaires

Nom du trait	Expression	Expression équivalente
`std::ops::Neg`	`-x`	`x.neg()`
`std::ops::Not`	`!x`	`x.not()`

Tous les types numériques signés de Rust implémentent `std::ops::Neg`, pour la négation unaireopérateur `-`; les types entiers et `bool` implement `std::ops::Not`, pour le complément unaireopérateur `!`. Il existe également des implémentations pour les références à ces types.

Notez que `!` complète les `bool` valeurs et effectue un complément au niveau du bit (c'est-à-dire inverse les bits) lorsqu'il est appliqué à des entiers ; il joue le rôle à la fois des opérateurs `!` et `~` de C et C++.

Les définitions de ces traits sont simples :

```
trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}
```

Nier un nombre complexe nie simplement chacun de ses composants. Voici comment nous pourrions écrire une implémentation générique de négation pour les `Complex` valeurs :

```
use std:: ops::Neg;

impl<T> Neg for Complex<T>
where
    T: Neg<Output = T>,
{
    type Output = Complex<T>;
    fn neg(self) -> Complex<T> {
        Complex {
            re: -self.re,
            im:-self.im,
        }
    }
}
```

Opérateurs binaires

Le binaire de Rustles opérateurs arithmétiques et au niveau du bit et leurs traits intégrés correspondants apparaissent dans le tableau 12-3 .

Tableau 12-3. Traits intégrés pour les opérateurs binaires

Catégorie	Nom du trait	Expression	Expression équivalente
Opérateurs arithmétiques	`std::ops::Add`	`x + y`	`x.add(y)`
	`std::ops::Sub`	`x - y`	`x.sub(y)`
	`std::ops::Mul`	`x * y`	`x.mul(y)`
	`std::ops::Div`	`x / y`	`x.div(y)`
	`std::ops::Rem`	`x % y`	`x.rem(y)`
Opérateurs au niveau du bit	`std::ops::BitAnd`	`x & y`	`x.bitand(y)`
	`std::ops::BitOr`	`x | y`	`x.bitor(y)`
	`std::ops::BitXor`	`x ^ y`	`x.bitxor(y)`
	`std::ops::Shl`	`x << y`	`x.shl(y)`
	`std::ops::Shr`	`x >> y`	`x.shr(y)`

Tous les types numériques de Rust implémentent les opérateurs arithmétiques. Les types entiers de Rust et `bool` implémentent les opérateurs au niveau du bit. Il existe également des implémentations qui acceptent les références à ces types comme opérandes ou les deux.

Tous les traits ont ici la même forme générale. La définition de `std::ops::BitXor`, pour l'`^` opérateur, ressemble à ceci :

```
trait BitXor<Rhs = Self> {
    type Output;
    fn bitxor(self, rhs: Rhs) -> Self::Output;
}
```

Au début de ce chapitre, nous avons également montré `std::ops::Add`, un autre trait de cette catégorie, ainsi que plusieurs exemples d'implémentations.

Vous pouvez utiliser l'`+` opérateurpour concaténer a `String` avec une `&str` tranche ou une autre `String`. Cependant, Rust ne permet pas à l'opérande gauche de `+` d'être un `&str`, pour décourager la construction de longues chaînes en concaténant à plusieurs reprises de petits morceaux sur la gauche. (Cela fonctionne mal, nécessitant un temps quadratique dans la longueur finale de la chaîne.) Généralement, la `write!` macroest meilleur pour construire des cordes morceau par morceau; nous montrons comment procéder dans « Ajout et insertion de texte » .

Opérateurs d'affectation composés

Un composéL'expression d'affectation est semblable à `x += y` ou `x &= y` : elle prend deux opérandes, effectue une opération sur eux comme une addition ou un ET au niveau du bit, et stocke le résultat dans l'opérande de gauche. Dans Rust, la valeur d'une expression d'affectation composée est toujours `()`, jamais la valeur stockée.

De nombreux langages ont des opérateurs comme ceux-ci et les définissent généralement comme des raccourcis pour des expressions telles que `x = x + y` ou `x = x & y`. Cependant, Rust n'adopte pas cette approche. Au lieu de cela, `x += y` est un raccourci pour l'appel de méthode `x.add_assign(y)`, où `add_assign` est la seule méthode du `std::ops::AddAssign` trait :

```
trait AddAssign<Rhs = Self> {
    fn add_assign(&mut self, rhs:Rhs);
}
```

Le tableau 12-4 montre tous les opérateurs d'affectation composés de Rust et les traits intégrés qui les implémentent.

Tableau 12-4. Caractéristiques intégrées pour les opérateurs d'affectation composés

Catégorie	Nom du trait	Expression	Expression équivalente
Opérateurs arithmétiques	`std::ops::AddAssign`	`x += y`	`x.add_assign(y)`
	`std::ops::SubAssign`	`x -= y`	`x.sub_assign(y)`
	`std::ops::MulAssign`	`x *= y`	`x.mul_assign(y)`
	`std::ops::DivAssign`	`x /= y`	`x.div_assign(y)`
	`std::ops::RemAssign`	`x %= y`	`x.rem_assign(y)`
Opérateurs au niveau du bit	`std::ops::BitAndAssign`	`x &= y`	`x.bitand_assign(y)`
	`std::ops::BitOrAssign`	`x |= y`	`x.bitor_assign(y)`
	`std::ops::BitXorAssign`	`x ^= y`	`x.bitxor_assign(y)`
	`std::ops::ShlAssign`	`x <<= y`	`x.shl_assign(y)`
	`std::ops::ShrAssign`	`x >>= y`	`x.shr_assign(y)`

Tous les types numériques de Rust implémentent les opérateurs d'affectation composés arithmétiques. Les types entiers de Rust et `bool` implémentent les opérateurs d'affectation composés au niveau du bit.

Une implémentation générique de `AddAssign` pour notre `Complex` type est simple :

```
use std:: ops::AddAssign;
```

```
impl<T> AddAssign for Complex<T>
where
    T: AddAssign<T>,
{
    fn add_assign(&mut self, rhs:Complex<T>) {
        self.re += rhs.re;
        self.im += rhs.im;
    }
}
```

Le trait intégré pour un opérateur d'affectation composé est complètement indépendant du trait intégré pour l'opérateur binaire correspondant. La mise en œuvre `std::ops::Add` n'implémente pas automatiquement `std::ops::AddAssign` ; si vous voulez que Rust autorise votre type comme opérande gauche d'un `+=` opérateur, vous devez vous `AddAssign` implémenter.

Comparaisons d'équivalence

L'égalité de Rustles opérateurs, `==` et `!=` , sont des raccourcis pour les appels aux `std::cmp::PartialEq` traits `eq` et aux `ne` méthodes :

```
assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));
```

Voici la définition de `std::cmp::PartialEq` :

```
trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) ->bool {
        !self.eq(other)
    }
}
```

Étant donné que la `ne` méthode a une définition par défaut, il vous suffit de définir `eq` pour implémenter le `PartialEq` trait, voici donc une implémentation complète pour `Complex` :

```
impl<T: PartialEq> PartialEq for Complex<T> {
    fn eq(&self, other: &Complex<T>) ->bool {
        self.re == other.re && self.im == other.im
```

```
    }
}
```

En d'autres termes, pour tout type de composant `T` qui peut lui-même être comparé pour l'égalité, cela implémente la comparaison pour `Complex<T>`. En supposant que nous ayons également implémenté `std::ops::Mul` quelque `Complex` part le long de la ligne, nous pouvons maintenant écrire :

```
let x = Complex { re: 5, im: 2 };
let y = Complex { re: 2, im: 5 };
assert_eq!(x * y, Complex { re: 0, im:29 });
```

Les implémentations de `PartialEq` sont presque toujours de la forme montrée ici : elles comparent chaque champ de l'opérande de gauche au champ correspondant de droite. Ceux-ci deviennent fastidieux à écrire, et l'égalité est une opération courante à prendre en charge, donc si vous le demandez, Rust générera `PartialEq` automatiquement une implémentation de pour vous. Ajoutez simplement à l' attribut `PartialEq` de la définition de type comme ceci : `derive`

```
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    ...
}
```

L'implémentation générée automatiquement de Rust est essentiellement identique à notre code écrit à la main, comparant tour à tour chaque champ ou élément du type. Rust peut également dériver `PartialEq` des implémentations pour `enum` les types. Naturellement, chacune des valeurs que le type contient (ou pourrait contenir, dans le cas d'un `enum`) doit elle-même implémenter `PartialEq`.

Contrairement aux traits arithmétiques et au niveau du bit, qui prennent leurs opérandes par valeur, `PartialEq` prend ses opérandes par référence. Cela signifie que la comparaison de non `Copy` -valeurs telles que `String` s, `Vec` s ou `HashMap` s ne les déplace pas, ce qui serait gênant :

```
let s = "d\x6fv\x65t\x61i\x6c".to_string();
let t = "\x64o\x76e\x74a\x69l".to_string();
assert!(s == t);  // s and t are only borrowed...

// ... so they still have their values here.
assert_eq!(format!("{} {}", s, t), "dovetail dovetail");
```

Cela nous amène à la limite du trait sur le `Rhs` paramètre de type, qui est d'un genre que nous n'avons jamais vu auparavant :

```
    where
        Rhs:?Sized,
```

Cela assouplit l'exigence habituelle de Rust selon laquelle les paramètres de type doivent être des types dimensionnés, nous permettant d'écrire des traits comme `PartialEq<str>` ou `PartialEq<[T]>` . Les méthodes `eq` et `ne` prennent des paramètres de type `&Rhs` , et comparer quelque chose avec a `&str` ou a `&[T]` est tout à fait raisonnable. Depuis `str` implements `PartialEq<str>` , les assertions suivantes sont équivalentes :

```
assert!("ungula" != "ungulate");
assert!("ungula".ne("ungulate"));
```

Ici, les deux `Self` et `Rhs` seraient le type non dimensionné `str` , faisant `ne` de `self` et `rhs` les paramètres les deux `&str` valeurs. Nous discuterons des types dimensionnés, des types non dimensionnés et du `Sized` trait en détail dans "Sized" .

Pourquoi appelle-t-on ce trait `PartialEq` ? La définition mathématique traditionnelle d'une *relation d'équivalence* , dont l'égalité est une instance, impose trois exigences. Pour toutes valeurs `x` et `y` :

- Si `x == y` est vrai, alors `y == x` doit être vrai aussi. En d'autres termes, l'échange des deux côtés d'une comparaison d'égalité n'affecte pas le résultat.
- Si `x == y` et `y == z` , alors ce doit être le cas que `x == z` . Étant donné n'importe quelle chaîne de valeurs, chacune égale à la suivante, chaque valeur de la chaîne est directement égale à toutes les autres. L'égalité est contagieuse.
- Il doit toujours être vrai que `x == x` .

Cette dernière exigence peut sembler trop évidente pour être énoncée, mais c'est exactement là que les choses tournent mal. Rust `f32` et `f64` sont des valeurs à virgule flottante standard IEEE. Selon cette norme, les expressions comme `0.0/0.0` et autres sans valeur appropriée doivent produire un *non-nombre spécial*valeurs, généralement appelées valeurs NaN. La norme exige en outre qu'une valeur NaN soit traitée comme inégale à toute autre valeur, y compris elle-même. Par exemple, la norme exige tous les comportements suivants :

```
assert!(f64::is_nan(0.0 / 0.0));
assert_eq!(0.0 / 0.0 == 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 != 0.0 / 0.0, true);
```

De plus, toute comparaison ordonnée avec une valeur NaN doit retourner false :

```
assert_eq!(0.0 / 0.0 < 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 > 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 <= 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 >= 0.0 / 0.0, false);
```

Ainsi, alors que l'opérateur de Rust `==` répond aux deux premières exigences pour les relations d'équivalence, il ne répond clairement pas à la troisième lorsqu'il est utilisé sur des valeurs à virgule flottante IEEE. C'est ce qu'on appelle une *relation d'équivalence partielle* , donc Rust utilise le nom du trait intégré de `PartialEq` l' `==` opérateur. Si vous écrivez du code générique avec des paramètres de type connus uniquement pour être `PartialEq` , vous pouvez supposer que les deux premières conditions sont remplies, mais vous ne devez pas supposer que les valeurs sont toujours égales à elles-mêmes.

Cela peut être un peu contre-intuitif et peut entraîner des bogues si vous n'êtes pas vigilant. Si vous préférez que votre code générique exige une relation d'équivalence complète, vous pouvez à la place utiliser le `std::cmp::Eq` trait comme une limite, qui représente une relation d'équivalence complète : si un type implémente `Eq` , alors `x == x` doit être `true` pour chaque valeur `x` de ce type. En pratique, presque tous les types qui implémentent `PartialEq` devraient `Eq` également implémenter ; `f32` et `f64` sont les seuls types de la bibliothèque standard qui sont, `PartialEq` mais pas `Eq` .

La bibliothèque standard définit `Eq` comme une extension de `PartialEq` , n'ajoutant aucune nouvelle méthode :

```
trait Eq:PartialEq<Self> {}
```

Si votre type est `PartialEq` et que vous souhaitez qu'il le soit `Eq` également, vous devez implémenter explicitement `Eq` , même si vous n'avez pas besoin de définir de nouvelles fonctions ou de nouveaux types pour le faire. La mise en œuvre `Eq` pour notre `Complex` type est donc rapide :

```
impl<T:Eq> Eq for Complex<T> {}
```

Nous pourrions l'implémenter encore plus succinctement en incluant simplement `Eq` dans l' `derive` attribut sur la `Complex` définition de type :

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    ...
}
```

Les implémentations dérivées sur un type générique peuvent dépendre des paramètres de type. Avec l' `derive` attribut, `Complex<i32>` implémenterait `Eq` , parce `i32` que , mais `Complex<f32>` implémenterait uniquement `PartialEq` , puisque `f32` n'implémente pas `Eq` .

Lorsque vous implémentez `std::cmp::PartialEq` vous-même, Rust ne peut pas vérifier que vos définitions pour les méthodes `eq` et `ne` se comportent réellement comme requis pour une équivalence partielle ou totale. Ils pourraient faire tout ce que vous voulez. Rust vous croit simplement sur parole que vous avez implémenté l'égalité d'une manière qui répond aux attentes des utilisateurs du trait.

Bien que la définition de `PartialEq` fournisse une définition par défaut pour `ne` , vous pouvez fournir votre propre implémentation si vous le souhaitez. Cependant, vous devez vous assurer que `ne` et `eq` sont des compléments exacts l'un de l'autre. Les utilisateurs du `PartialEq` trait supposeront qu'il en est ainsi.

Comparaisons ordonnées

Rust spécifie le comportement de la commandeopérateurs de comparaison `<` , `>` , `<=` , et `>=` tous en termes d'un seul trait, `std::cmp::PartialOrd` :

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) ->Option<Ordering>;

    fn lt(&self, other: &Rhs) -> bool { ... }
    fn le(&self, other: &Rhs) -> bool { ... }
    fn gt(&self, other: &Rhs) -> bool { ... }
    fn ge(&self, other: &Rhs) ->bool { ... }
}
```

Notez que `PartialOrd<Rhs>` s'étend `PartialEq<Rhs>` : vous ne pouvez effectuer des comparaisons ordonnées que sur des types que vous pouvez également comparer pour l'égalité.

La seule méthode que `PartialOrd` vous devez implémenter vous-même est `partial_cmp`. Lorsque `partial_cmp` renvoie `Some(o)`, alors `o` indique `self` la relation de `other` :

```
enum Ordering {
    Less,       // self < other
    Equal,      // self == other
    Greater,    // self > other
}
```

Mais si `partial_cmp` renvoie `None`, cela signifie que `self` et `other` ne sont pas ordonnés l'un par rapport à l'autre : aucun n'est plus grand que l'autre, ni égal. Parmi tous les types primitifs de Rust, seules les comparaisons entre des valeurs à virgule flottante sont renvoyées `None` : en particulier, la comparaison d'une valeur NaN (pas un nombre) avec n'importe quoi d'autre renvoie `None`. Nous donnons plus d'informations sur les valeurs de NaN dans "Comparaisons d'équivalence".

Comme les autres opérateurs binaires, pour comparer des valeurs de deux types `Left` et `Right`, `Left` il faut implémenter `PartialOrd<Right>`. Des expressions telles que `x < y` ou `x >= y` sont des raccourcis pour les appels de `PartialOrd` méthodes, comme indiqué dans le Tableau 12-5.

Tableau 12-5. Opérateurs et `PartialOrd` méthodes de comparaison ordonnée

Expression	Appel de méthode équivalent	Définition par défaut
`x < y`	`x.lt(y)`	`x.partial_cmp(&y) == Some(Less)`
`x > y`	`x.gt(y)`	`x.partial_cmp(&y) == Some(Greater)`
`x <= y`	`x.le(y)`	`matches!(x.partial_cmp(&y), Some(Less | Equal))`
`x >= y`	`x.ge(y)`	`matches!(x.partial_cmp(&y), Some(Greater | Equal))`

Comme dans les exemples précédents, le code d'appel de méthode équivalent indiqué suppose que `std::cmp::PartialOrd` et `std::cmp::Ordering` sont dans la portée.

Si vous savez que les valeurs de deux types sont toujours ordonnées l'une par rapport à l'autre, vous pouvez implémenter le `std::cmp::Ord` trait plus strict :

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) ->Ordering;
}
```

La `cmp` méthode ici renvoie simplement un `Ordering` , au lieu d'un `Option<Ordering>` like `partial_cmp` : `cmp` déclare toujours ses arguments égaux ou indique leur ordre relatif. Presque tous les types qui implémentent `PartialOrd` devraient également implémenter `Ord` . Dans la bibliothèque standard, `f32` et `f64` sont les seules exceptions à cette règle.

Puisqu'il n'y a pas d'ordre naturel sur les nombres complexes, nous ne pouvons pas utiliser notre `Complex` type des sections précédentes pour montrer un exemple d'implémentation de `PartialOrd` . Au lieu de cela, supposons que vous travaillez avec le type suivant, représentant l'ensemble des nombres compris dans un intervalle semi-ouvert donné :

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
```

```
    lower: T, // inclusive
    upper:T, // exclusive
}
```

Vous aimeriez que les valeurs de ce type soient partiellement ordonnées : un intervalle est inférieur à un autre s'il tombe entièrement avant l'autre, sans chevauchement. Si deux intervalles inégaux se chevauchent, ils ne sont pas ordonnés : un élément de chaque côté est inférieur à un élément de l'autre. Et deux intervalles égaux sont simplement égaux. L'implémentation suivante de `PartialOrd` implémente ces règles :

```
use std:: cmp::{Ordering, PartialOrd};

impl<T: PartialOrd> PartialOrd<Interval<T>> for Interval<T> {
    fn partial_cmp(&self, other: &Interval<T>) -> Option<Ordering> {
        if self == other {
            Some(Ordering:: Equal)
        } else if self.lower >= other.upper {
            Some(Ordering:: Greater)
        } else if self.upper <= other.lower {
            Some(Ordering::Less)
        } else {
            None
        }
    }
}
```

Avec cette implémentation en place, vous pouvez écrire ce qui suit :

```
assert!(Interval { lower: 10, upper: 20 } <  Interval { lower: 20, upper: 4
assert!(Interval { lower: 7,  upper: 8  } >= Interval { lower: 0,  upper: 1
assert!(Interval { lower: 7,  upper: 8  } <= Interval { lower: 7,  upper:8

// Overlapping intervals aren't ordered with respect to each other.
let left  = Interval { lower: 10, upper: 30 };
let right = Interval { lower: 20, upper:40 };
assert!(!(left < right));
assert!(!(left >= right));
```

Alors que `PartialOrd` c'est ce que vous verrez habituellement, les classements totaux définis avec `Ord` sont nécessaires dans certains cas, comme les méthodes de tri implémentées dans la bibliothèque standard. Par exemple, le tri des intervalles n'est pas possible avec seulement une `PartialOrd` implémentation. Si vous voulez les trier, vous devrez combler les vides des caisses non ordonnées. Vous voudrez peut-être trier par limite supérieure, par exemple, et c'est facile de le faire avec `sort_by_key` :

```
intervals.sort_by_key(|i| i.upper);
```

Le `Reverse` type wrapper en profite en implémentant `Ord` une méthode qui inverse simplement tout ordre. Pour tout type `T` qui implémente `Ord`, `std::cmp::Reverse<T>` implémente `Ord` aussi, mais avec un ordre inversé. Par exemple, trier nos intervalles de haut en bas par borne inférieure est simple:

```
use std:: cmp::Reverse;
intervals.sort_by_key(|i| Reverse(i.lower));
```

Index et IndexMut

Vous pouvez spécifier comment une indexationexpression like `a[i]` fonctionne sur votre type en implémentant les traits `std::ops::Index` et . `std::ops::IndexMut` Les tableaux prennent `[]` directement en charge l'opérateur, mais sur tout autre type, l'expression `a[i]` est normalement un raccourci pour `*a.index(i)`, où `index` est une méthode du `std::ops::Index` trait. Cependant, si l'expression est affectée ou empruntée de manière mutable, il s'agit plutôt d'un raccourci pour `*a.index_mut(i)`, un appel à la méthode du `std::ops::IndexMut` trait.

Voici les définitions des traits :

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

Notez que ces traits prennent le type de l'expression d'index comme paramètre. Vous pouvez indexer une tranche avec un seul `usize`, faisant référence à un seul élément, car les tranches implémentent `Index<usize>`. Mais vous pouvez faire référence à une sous-tranche avec une expression comme `a[i..j]` parce qu'ils implémentent également `Index<Range<usize>>`. Cette expression est un raccourci pour :

```
*a.index(std:: ops:: Range { start: i, end:j })
```

Les collections Rust `HashMap` et `BTreeMap` vous permettent d'utiliser n'importe quel type hachable ou ordonné comme index. Le code suivant fonctionne car `HashMap<&str, i32>` implémente `Index<&str>` :

```
use std:: collections:: HashMap;
let mut m = HashMap::new();
m.insert("十", 10);
m.insert("百", 100);
m.insert("千", 1000);
m.insert("万", 1_0000);
m.insert("億", 1_0000_0000);

assert_eq!(m["十"], 10);
assert_eq!(m["千"], 1000);
```

Ces expressions d'indexation sont équivalentes à :

```
use std:: ops::Index;
assert_eq!(*m.index("十"), 10);
assert_eq!(*m.index("千"), 1000);
```

Le `Index` type associé au trait `Output` spécifie le type produit par une expression d'indexation : pour notre `HashMap` , le type `Index` de l'implémentation `Output` est `i32` .

Le `IndexMut` trait s'étend `Index` avec une `index_mut` méthode qui prend une référence mutable à `self` et renvoie une référence mutable à une `Output` valeur. Rust sélectionne automatiquement `index_mut` lorsque l'expression d'indexation se produit dans un contexte où cela est nécessaire. Par exemple, supposons que nous écrivions ce qui suit :

```
let mut desserts =
    vec!["Howalon".to_string(), "Soan papdi".to_string()];
desserts[0].push_str(" (fictional)");
desserts[1].push_str(" (real)");
```

Étant donné que la `push_str` méthode fonctionne sur `&mut self` , ces deux dernières lignes sont équivalentes à :

```
use std:: ops::IndexMut;
(*desserts.index_mut(0)).push_str(" (fictional)");
(*desserts.index_mut(1)).push_str(" (real)");
```

Une limitation de `IndexMut` est que, de par sa conception, il doit renvoyer une référence mutable à une certaine valeur. C'est pourquoi vous ne pouvez pas utiliser une expression comme `m["十"] = 10;` pour insérer une valeur dans le `HashMap` `m` : la table devrait d' `"十"` abord créer une entrée, avec une valeur par défaut, et renvoyer une référence mutable à celle-ci. Mais tous les types n'ont pas de valeurs par défaut bon marché, et certains peuvent être coûteux à supprimer ; ce serait un gaspillage de créer une telle valeur pour être immédiatement abandonnée par l'affectation. (Il est prévu d'améliorer cela dans les versions ultérieures du langage.)

L'utilisation la plus courante de l'indexation concerne les collections. Par exemple, supposons que nous travaillions avec des images bitmap, comme celles que nous avons créées dans le traceur d'ensemble de Mandelbrot au chapitre 2 . Rappelez-vous que notre programme contenait un code comme celui-ci :

```
pixels[row * bounds.0 + column] = ...;
```

Il serait plus agréable d'avoir un `Image<u8>` type qui agit comme un tableau à deux dimensions, nous permettant d'accéder aux pixels sans avoir à écrire toute l'arithmétique :

```
image[row][column] = ...;
```

Pour ce faire, nous devrons déclarer une structure :

```
struct Image<P> {
    width: usize,
    pixels:Vec<P>,
}

impl<P: Default + Copy> Image<P> {
    /// Create a new image of the given size.
    fn new(width: usize, height: usize) -> Image<P> {
        Image {
            width,
            pixels: vec![P::default(); width * height],
        }
    }
}
```

Et voici des implémentations de `Index` et `IndexMut` qui feraient l'affaire:

```
impl<P> std:: ops:: Index<usize> for Image<P> {
    type Output = [P];
    fn index(&self, row: usize) ->&[P] {
        let start = row * self.width;
        &self.pixels[start..start + self.width]
    }
}

impl<P> std:: ops:: IndexMut<usize> for Image<P> {
    fn index_mut(&mut self, row: usize) ->&mut [P] {
        let start = row * self.width;
        &mut self.pixels[start..start + self.width]
    }
}
```

Lorsque vous indexez dans un `Image`, vous récupérez une tranche de pixels ; l'indexation de la tranche vous donne un pixel individuel.

Notez que lorsque nous écrivons `image[row][column]`, si `row` est hors limites, notre `.index()` méthode essaiera d'indexer `self.pixels` hors limites, déclenchant une panique. C'est ainsi `Index` que `IndexMut` les implémentations sont censées se comporter : un accès hors limites est détecté et provoque une panique, comme lorsque vous indexez un tableau, une tranche ou un vecteur hors limites.

Autres opérateurs

Pas toutles opérateurs peuvent être surchargés dans Rust. Depuis Rust 1.56, l' `?` opérateur de vérification des erreurs ne fonctionne qu'avec `Result` et quelques autres types de bibliothèques standard, mais des travaux sont en cours pour l'étendre également aux types définis par l'utilisateur. De même, les opérateurs logiques `&&` et `||` sont limités aux valeurs booléennes uniquement. Les opérateurs `..` et `..=` créent toujours une structure représentant les limites de la plage, l' `&` opérateur emprunte toujours des références et l' `=` opérateur déplace ou copie toujours des valeurs. Aucun d'entre eux ne peut être surchargé.

L'opérateur de déréférencement `*val`, et l'opérateur point pour accéder aux champs et aux méthodes d'appel, comme dans `val.field` et `val.method()`, peuvent être surchargés à l'aide des traits `Deref` et `DerefMut`, qui sont traités dans le chapitre suivant. (Nous ne les avons pas inclus ici car ces traits font plus que simplement surcharger quelques opérateurs.)

Rust ne prend pas en charge la surcharge de l'opérateur d'appel de fonction, `f(x)` . Au lieu de cela, lorsque vous avez besoin d'une valeur appelable, vous écrirez généralement simplement une fermeture. Nous expliquerons comment cela fonctionne et aborderons les traits spéciaux `Fn` , `FnMut` et au chapitre 14 . `FnOnce`

1 Les programmeurs Lisp se réjouissent ! L'expression `<i32 as Add>::add` est l' `+` opérateur sur `i32` , capturé en tant que valeur de fonction.