Chapitre 12. Surcharge de l'opérateur

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

```
#[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 ajouter et multiplier des nombres comme n'importe quel type numérique intégré, en utilisant rust et les opérateurs: `Complex + *`

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

Vous pouvez également faire en sorte que vos propres types prennent en charge l'arithmétique et d'autres opérateurs, simplement en implémentant quelques traits intégrés. *C'est ce qu'on appelle la surcharge d'opérateur*, et l'effet est un peu similaire à la surcharge d'opérateur en C ++, C #, Python et Ruby.

Les caractéristiques de surcharge de l'opérateur se répartissent en quelques catégories en fonction de la partie de la langue qu'ils prennent en charge, comme le montre 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 à points décrite dans « Reverse-Engineering Bounds » qui fonctionnent sur les types les plus naturellement utilisés 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 la surcharge de l'opérateur

Catégorie	Trait	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 binaires	`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`

Catégorie	Trait	Opérateur
Opérateurs arithmétiques d'affectation composée	`std::ops::AddAssign`	`x += y`
	`std::ops::SubAssign`	`x -= y`
	`std::ops::MulAssign`	`x *= y`
	`std::ops::DivAssign`	`x /= y`
	`std::ops::RemAssign`	`x %= y`
Opérateurs binaires 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`
Indexation	`std::ops::Index`	`x[y], &x[y]`

Catégorie	Trait	Opérateur
	`std::ops::IndexMut`	`x[y] = z, &mut x[y]`

Opérateurs arithmétiques et binaires

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

Si vous voulez essayer d'écrire, vous devrez mettre le trait dans la portée afin que sa méthode soit visible. Cela fait, vous pouvez traiter toute l'arithmétique comme des appels de fonction: `z.add(c) Add` [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 est la capacité d'ajouter une valeur à vous-même. Par exemple, si vous souhaitez pouvoir ajouter des valeurs à votre type, votre type doit implémenter à la fois et . Le paramètre de type du trait est défini par défaut sur , 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'ajout. `Add<T> T i32 u32 Add<i32> Add<u32> Rhs Self Add Output`

Par exemple, pour pouvoir additionner des valeurs, il faut implémenter . Puisque nous ajoutons un type à lui-même, nous écrivons simplement : `Complex<i32> Complex<i32> Add<Complex<i32>> 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 séparément pour , , , et ainsi de suite. Toutes les définitions seraient exactement les mêmes, 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'ajout: `Add Complex<i32> Complex<f32> Complex<f64>`

```
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 , nous nous limitons aux types qui peuvent être ajoutés à eux-mêmes, donnant une autre valeur. C'est une restriction raisonnable, mais nous pourrions assouplir encore les choses : le trait n'exige pas que les deux opérandes aient le même type, ni ne limite le type de résultat. Ainsi, une implémentation générique maximale permettrait aux opérandes gauche et droit de varier indépendamment et de produire une valeur de

n'importe quel type de composant produit par l'addition : `where T: Add<Output=T> T T Add + Complex`

```
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,
        }
    }
}
```

Dans la pratique, cependant, Rust a tendance à éviter de prendre en charge des opérations de type mixte. Puisque notre paramètre de type doit implémenter , il s'ensuit généralement que et va être le même type: il n'y a tout simplement pas beaucoup de types disponibles pour cette implémentation autre chose. Donc, en fin de compte, cette version générique maximale peut ne pas être beaucoup plus utile que la définition générique précédente, plus simple. `L Add<R> L R L`

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 composée. Au sein de chaque groupe, les traits et leurs méthodes ont tous la même forme, nous allons donc couvrir un exemple de chacun.

Opérateurs unaires

Mis à part l'opérateur de déréférencement , que nous couvrirons séparément dans « Deref et DerefMut », Rust dispose de deux opérateurs unaires que vous pouvez personnaliser, illustrés dans le tableau 12-2. *

Tableau 12-2. Caractéristiques intégrées 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 , pour l'opérateur de négation unaire ; les types entiers et implémenter , pour l'opérateur de complément unaire . Il existe également des implémentations pour les références à ces types. `std::ops::Neg - bool std::ops::Not !`

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

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;
}
```

La négation d'un nombre complexe annule simplement chacun de ses composants. Voici comment nous pourrions écrire une implémentation générique de la négation pour les valeurs : `Complex`

```
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

Les opérateurs arithmétiques binaires et binaires de Rust et leurs caractéristiques intégrées correspondantes apparaissent dans le tableau 12-3.

Tableau 12-3. Caractéristiques intégrées 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 binaires	`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 implémentent les opérateurs binaires. Il existe également des implémentations qui acceptent les références à ces types en tant qu'opérandes ou les deux. `bool`

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

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

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

Vous pouvez utiliser l'opérateur pour concaténer a avec une tranche ou un autre . Cependant, Rust ne permet pas à l'opérande gauche d'être un , pour décourager la construction de longues cordes 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 macro est meilleure pour construire des cordes pièce par pièce; nous montrons comment faire cela dans « Ajout et insertion de texte ». `+ String &str String + &str write!`

Opérateurs d'affectation composée

Une expression d'affectation composée est une expression semblable ou : elle prend deux opérandes, effectue une opération sur eux comme l'addition ou un AND binaire, et stocke le résultat dans l'opérande gauche. Dans Rust, la valeur d'une expression d'affectation composée est toujours , jamais la valeur stockée. `x += y x &= y ()`

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

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

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

Tableau 12-4. Caractéristiques intégrées pour les opérateurs d'affectation de 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 binaires	`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 composée arithmétique. Les types entiers de Rust et implémentent les opérateurs d'affectation composée binaire. `bool`

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

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

Comparaisons d'équivalence

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

```
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 méthode a une définition par défaut, il vous suffit de définir pour implémenter le trait, voici donc une implémentation complète pour : `ne eq PartialEq 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 qui peut lui-même être comparé pour l'égalité, cela implémente la comparaison pour . En supposant que nous ayons également implémenté quelque part le long de la ligne, nous pouvons maintenant
écrire: `T Complex<T> std::ops::Mul Complex`

```
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 sont presque toujours de la forme montrée ici: elles comparent chaque champ de l'opérande 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 demandez, Rust générera automatiquement une implémentation de pour vous. Ajoutez simplement à l'attribut de la définition de type comme suit
: `PartialEq PartialEq PartialEq derive`

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

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

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

```
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 liaison du trait sur le paramètre type, qui est d'un type que nous n'avons jamais vu auparavant: `Rhs`

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

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

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

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

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

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

Cette dernière exigence peut sembler trop évidente pour valoir la peine d'être énoncée, mais c'est exactement là que les choses tournent mal. Rust et sont des valeurs à virgule flottante standard IEEE. Selon cette norme, les expressions comme et d'autres sans valeur appropriée doivent produire des valeurs spéciales *non numériques*, généralement appelées

valeurs NaN. La norme exige en outre qu'une valeur NaN soit traitée comme inégale à toutes les autres valeurs, y compris elle-même. Par exemple, la norme requiert tous les comportements suivants
: `f32 f64 0.0/0.0`

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

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

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

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

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

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

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

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

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

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

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

Comparaisons ordonnées

Rust spécifie le comportement des opérateurs de comparaison ordonnés , , , et le tout 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 { ... }
}
```

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

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

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

Mais si les retours , cela signifie et sont non ordonnés les uns par rapport aux autres: ni l'un ni l'autre n'est plus grand que l'autre, ni égal. Parmi tous les types primitifs de Rust, seules les comparaisons entre les valeurs en virgule flottante reviennent : plus précisément, la comparaison d'une valeur NaN (pas un nombre) avec quoi que ce soit d'autre renvoie. Nous donnons plus de détails sur les valeurs NaN dans « Comparaisons d'équivalence ». `partial_cmp None self other None None`

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

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

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 affiché suppose que et sont dans la portée. `std::cmp::PartialOrd` `std::cmp::Ordering`

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

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

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

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

```
#[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // inclusive
    upper: T, // exclusive
}
```

Vous souhaitez 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 tout simplement égaux. La mise en œuvre suivante de ces règles : `PartialOrd`

```
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
        }
    }
}
```

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

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

// 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));
```

Bien que ce soit ce que vous verrez habituellement, les ordres totaux définis avec sont nécessaires dans certains cas, tels que les méthodes de tri implémentées dans la bibliothèque standard. Par exemple, les intervalles de tri ne sont pas possibles avec seulement une implémentation. Si vous

voulez les trier, vous devrez combler les lacunes des cas non ordonnés. Vous voudrez peut-être trier par limite supérieure, par exemple, et il est facile de le faire avec : `PartialOrd Ord PartialOrd sort_by_key`

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

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

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

Index et IndexMut

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

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 , faisant référence à un seul élément, car les tranches implémentent . Mais vous pouvez vous référer à une sous-section avec une expression comme parce

qu'ils implémentent également . Cette expression est un raccourci pour : `usize Index<usize> a[i..j] Index<Range<usize>>`

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

Rust's et collections vous permettent d'utiliser n'importe quel type hashable ou ordonné comme index. Le code suivant fonctionne car implémente : `HashMap BTreeMap HashMap<&str, i32> 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 type associé au trait spécifie le type produit par une expression d'indexation : pour notre , le type de l'implémentation est . `Index Output HashMap Index Output i32`

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

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

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

L'une des limites est que, de par sa conception, il doit renvoyer une référence modifiable à une certaine valeur. C'est pourquoi vous ne pouvez pas utiliser une expression comme insérer une valeur dans le : la table devrait créer une entrée pour d'abord, avec une valeur par défaut, et renvoyer une référence modifiable à cela. Mais tous les types n'ont pas de valeurs par défaut bon marché, et certains peuvent être coûteux à abandonner; ce serait un gaspillage de créer une telle valeur pour être immédiatement abandonné par la cession. (Il est prévu d'améliorer cela dans les versions ultérieures de la langue.) `IndexMut m["十"] = 10; HashMap m "十"`

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 Mandelbrot au chapitre 2. Rappelons que notre programme contenait du code comme celui-ci :

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

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

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

Pour ce faire, nous devrons déclarer une struct:

```
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 les implémentations de et cela correspondrait à la facture: `Index IndexMut`

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

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

Autres opérateurs

Tous les opérateurs ne peuvent pas être surchargés dans Rust. À partir de Rust 1.56, l'opérateur de vérification des erreurs ne fonctionne qu'avec quelques autres types de bibliothèque 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. `? Result && || .. ..= & =`

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

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

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