Chapitre 18. Entrées et sorties

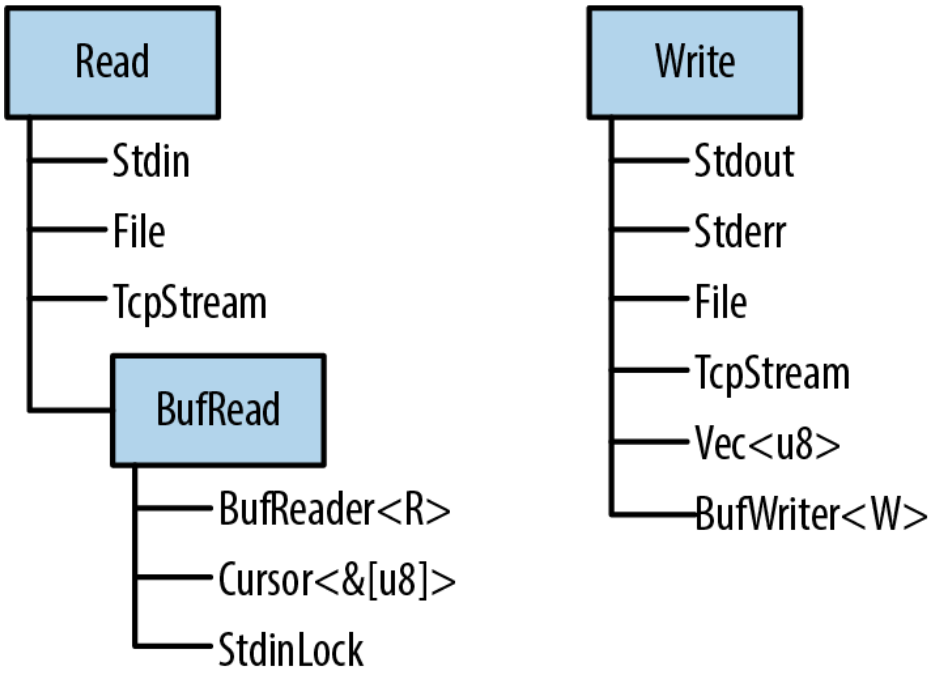
> *Doolittle: Quelles preuves concrètes avez-vous que vous existez?*
>
> *Bombe #20: Hmmmm... puits... Je pense, donc je le suis.*
>
> *Doolittle : C'est bien. C'est très bien. Mais comment savez-vous que quelque chose d'autre existe?*
>
> *Bombe #20 : Mon appareil sensoriel me le révèle.*
>
> —*Étoile noire*

Les fonctionnalités de bibliothèque standard de Rust pour l'entrée et la sortie sont organisées autour de trois traits, , , et : `Read` `BufRead` `Write`

- Les valeurs implémentées ont des méthodes pour l'entrée orientée octet. On les appelle des *lecteurs*. `Read`
- Les valeurs *implémentées sont des lecteurs mis en mémoire tampon*. Ils prennent en charge toutes les méthodes de , plus les méthodes de lecture de lignes de texte et ainsi de suite. `BufRead` `Read`
- Les valeurs implémentées prennent en charge à la fois la sortie de texte orientée octet et UTF-8. On les appelle *des écrivains*. `Write`

La figure 18-1 montre ces trois traits et quelques exemples de types de lecteurs et d'écrivains.

Dans ce chapitre, nous expliquerons comment utiliser ces traits et leurs méthodes, couvrirons les types de lecteur et d'écrivain illustrés dans la figure et montrerons d'autres façons d'interagir avec les fichiers, le terminal et le réseau.



Graphique 18-1. Les trois principaux traits d'E/S de Rust et les types sélectionnés qui les implémentent

Lecteurs et écrivains

Les lecteurs sont des valeurs à partir desquelles votre programme peut lire des octets. En voici quelques exemples :

- Fichiers ouverts à l'aide de `std::fs::File::open(filename)`
- `std::net::TcpStream` s, pour recevoir des données sur le réseau
- `std::io::stdin()` , pour la lecture à partir du flux d'entrée standard du processus
- `std::io::Cursor<&[u8]>` et les valeurs, qui sont des lecteurs qui « lisent » à partir d'un tableau d'octets ou d'un vecteur qui est déjà en mémoire `std::io::Cursor<Vec<u8>>`

Les rédacteurs sont des valeurs sur lesquelles votre programme peut écrire des octets. En voici quelques exemples :

- Fichiers ouverts à l'aide de `std::fs::File::create(filename)`
- `std::net::TcpStream` s, pour l'envoi de données sur le réseau
- `std::io::stdout()` et , pour écrire au terminal `std::io:stderr()`
- `Vec<u8>` , un écrivain dont les méthodes s'ajoutent au vecteur `write`
- `std::io::Cursor<Vec<u8>>` , qui est similaire mais vous permet à la fois de lire et d'écrire des données, et de chercher à différentes positions dans le vecteur
- `std::io::Cursor<&mut [u8]>` , qui ressemble beaucoup à , sauf qu'il ne peut pas faire croître la mémoire tampon, car il ne s'agit que d'une tranche d'un tableau d'octets existant `std::io::Cursor<Vec<u8>>`

Comme il existe des traits standard pour les lecteurs et les écrivains (et), il est assez courant d'écrire du code générique qui fonctionne sur une variété de canaux d'entrée ou de sortie. Par exemple, voici une fonction qui copie tous les octets de n'importe quel lecteur vers n'importe quel écrivain : `std::io::Read std::io::Write`

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

const DEFAULT_BUF_SIZE: usize = 8 * 1024;

pub fn copy<R: ?Sized, W: ?Sized>(reader: &mut R, writer: &mut W)
    -> io::Result<u64>
    where R: Read, W: Write
{
    let mut buf = [0; DEFAULT_BUF_SIZE];
    let mut written = 0;
    loop {
        let len = match reader.read(&mut buf) {
            Ok(0) => return Ok(written),
            Ok(len) => len,
            Err(ref e) if e.kind() == ErrorKind::Interrupted => continu
            Err(e) => return Err(e),
        };
        writer.write_all(&buf[..len])?;
        written += len as u64;
    }
}
```

Il s'agit de l'implémentation de la bibliothèque standard de Rust. Comme il est générique, vous pouvez l'utiliser pour copier des données de a à a , de à un en mémoire , etc. `std::io::copy() File TcpStream Stdin Vec<u8>`

Si le code de gestion des erreurs ici n'est pas clair, revenez au chapitre 7. Nous utiliserons le type constamment dans les pages à venir; il est important d'avoir une bonne compréhension de son fonctionnement. `Result`

Les trois traits , , et , avec , sont si couramment utilisés qu'il existe un module contenant uniquement ces traits: `std::io Read BufRead Write Seek prelude`

```
use std::io::prelude::*;
```

Vous le verrez une ou deux fois dans ce chapitre. Nous prenons également l'habitude d'importer le module lui-même: `std::io`

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

Le mot-clé ici est déclaré comme alias pour le module. De cette façon, et peut être écrit de manière plus concise au fur et à mesure, et ainsi de suite. `self io std::io std::io::Result std::io::Error io::Result io::Error`

Lecteurs

`std::io::Read` dispose de plusieurs méthodes de lecture des données. Tous prennent le lecteur lui-même par référence. `mut`

`reader.read(&mut buffer)`

Lit certains octets de la source de données et les stocke dans le fichier . Le type de l'argument est . Cela se lit jusqu'à des octets. `buffer buffer &mut [u8] buffer.len()`

Le type de retour est , qui est un alias de type pour . En cas de réussite, la valeur est le nombre d'octets lus, qui peut être égal ou inférieur à , *même s'il y a plus de données à venir,* au gré de la source de données. signifie qu'il n'y a plus d'entrée à lire. `io::Result<u64> Result<u64, io::Error> u64 buffer.len() Ok(0)`

En cas d'erreur, renvoie , où est une valeur. An est imprimable, pour le bénéfice des humains; pour les programmes, il dispose d'une méthode qui renvoie un code d'erreur de type . Les membres de cet enum ont des noms comme et . La plupart indiquent des erreurs graves qui ne peuvent pas être ignorées, mais un type d'erreur doit être traité spécialement. correspond au code d'erreur Unix , ce qui signifie que la lecture a été interrompue par un signal. À moins que le programme ne soit conçu pour faire quelque chose d'intelligent avec les signaux, il devrait simplement réessayer la lecture. Le code de , dans la section précédente, en montre un exemple. `.read() Err(err) err io::Error io::Error .kind() io::ErrorKind PermissionDenied ConnectionReset io::ErrorKind::Interrupted EINTR copy()`

Comme vous pouvez le voir, la méthode est de très bas niveau, héritant même des bizarreries du système d'exploitation sous-jacent. Si vous implémentez le trait pour un nouveau type de source de données, cela vous donne beaucoup de marge de manœuvre. Si vous essayez de lire des données, c'est pénible. Par conséquent, Rust fournit plusieurs méthodes de commodité de niveau supérieur. Tous ont des implémentations par défaut en termes de . Ils gèrent tous , donc vous n'avez pas à le faire. `.read()` `Read` `.read()` `ErrorKind::Interrupted`

`reader.read_to_end(&mut byte_vec)`

Lit toutes les entrées restantes de ce lecteur, en l'ajoutant à , qui est un fichier . Renvoie un , le nombre d'octets lus. `byte_vec` `Vec<u8>` `io::Result<usize>`

Il n'y a pas de limite à la quantité de données que cette méthode empilera dans le vecteur, alors ne l'utilisez pas sur une source non fiable. (Vous pouvez imposer une limite à l'aide de la méthode décrite dans la liste suivante.) `.take()`

`reader.read_to_string(&mut string)`

C'est la même chose, mais ajoute les données au fichier . Si le flux n'est pas valide UTF-8, cela renvoie une erreur. `String` `ErrorKind::InvalidData`

Dans certains langages de programmation, l'entrée d'octets et l'entrée de caractères sont gérées par différents types. De nos jours, UTF-8 est si dominant que Rust reconnaît cette norme de facto et prend en charge UTF-8 partout. D'autres jeux de caractères sont pris en charge avec la caisse open source. `encoding`

`reader.read_exact(&mut buf)`

Lit exactement suffisamment de données pour remplir le tampon donné. Le type d'argument est . Si le lecteur manque de données avant de lire des octets, cela renvoie une erreur. `&[u8]` `buf.len()` `ErrorKind::UnexpectedEof`

Ce sont les principales méthodes du trait. En outre, il existe trois méthodes d'adaptateur qui prennent la valeur by, la transformant en un itérateur ou un lecteur différent : `Read` `reader`

`reader.bytes()`

Renvoie un itérateur sur les octets du flux d'entrée. Le type d'élément est , de sorte qu'une vérification d'erreur est requise pour chaque octet. De plus, cela appelle une fois par octet, ce qui sera très inefficace si le lecteur n'est pas mis en mémoire tampon. `io::Result<u8> reader.read()`

`reader.chain(reader2)`

Renvoie un nouveau lecteur qui produit toutes les entrées de , suivies de toutes les entrées de . `reader reader2`

`reader.take(n)`

Renvoie un nouveau lecteur qui lit à partir de la même source que , mais qui est limité aux octets d'entrée. `reader n`

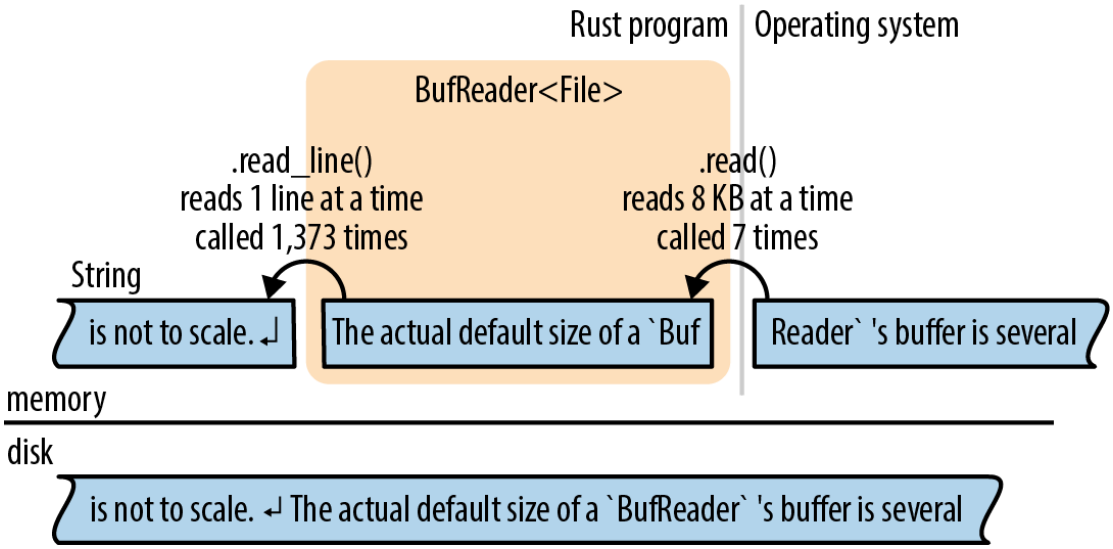
Il n'existe aucune méthode pour fermer un lecteur. Les lecteurs et les rédacteurs implémentent généralement de sorte qu'ils sont fermés automatiquement. `Drop`

Lecteurs tamponnés

Pour plus d'efficacité, les lecteurs et les rédacteurs peuvent être *mis en mémoire tampon*, ce qui signifie simplement qu'ils ont un morceau de mémoire (un tampon) qui contient des données d'entrée ou de sortie en mémoire. Cela permet d'économiser sur les appels système, comme illustré à la figure 18-2. L'application lit les données du , dans cet exemple en appelant sa méthode. Le à son tour obtient son entrée en plus gros morceaux du système d'exploitation. `BufReader .read_line() BufReader`

Cette image n'est pas à l'échelle. La taille par défaut réelle de la mémoire tampon d'un tampon est de plusieurs kilo-octets, de sorte qu'un seul système peut servir des centaines d'appels. Cela est important car les appels système sont lents. `BufReader read .read_line()`

(Comme le montre l'image, le système d'exploitation dispose également d'une mémoire tampon, pour la même raison : les appels système sont lents, mais la lecture des données d'un disque est plus lente.)



Graphique 18-2. Un lecteur de fichiers mis en mémoire tampon

Les lecteurs mis en mémoire tampon implémentent les deux et un deuxième trait, , qui ajoute les méthodes suivantes : `Read BufRead`

`reader.read_line(&mut line)`

Lit une ligne de texte et l'ajoute à , qui est un fichier . Le caractère de nouvelle ligne à la fin de la ligne est inclus dans . Si l'entrée a des terminaisons de ligne de style Windows, , les deux caractères sont inclus dans . `line String '\n' line "\r\n" line`

La valeur renvoyée est un , le nombre d'octets lus, y compris la fin de la ligne, le cas échéant. `io::Result<usize>`

Si le lecteur est à la fin de l'entrée, cela laisse inchangé et renvoie `. line Ok(0)`

`reader.lines()`

Renvoie un itérateur sur les lignes de l'entrée. Le type d'élément est . Les caractères de nouvelle ligne *ne sont pas* inclus dans les chaînes. Si l'entrée a des terminaisons de ligne de style Windows, les deux caractères sont supprimés. `io::Result<String> "\r\n"`

Cette méthode est presque toujours ce que vous voulez pour la saisie de texte. Les deux sections suivantes montrent quelques exemples de son utilisation.

`reader.read_until(stop_byte, &mut byte_vec)`, *`reader.split(stop_byte)`*

Ceux-ci sont comme et , mais orientés octets, produisant s au lieu de s. Vous choisissez le délimiteur

`. .read_line() .lines() Vec<u8> String stop_byte`

`BufRead` fournit également une paire de méthodes de bas niveau et , pour un accès direct à la mémoire tampon interne du lecteur. Pour en savoir plus sur ces méthodes, consultez la documentation en ligne. `.fill_buf()` `.consume(n)`

Les deux sections suivantes couvrent plus en détail les lecteurs tamponnés.

Lignes de lecture

Voici une fonction qui implémente l'utilitaire Unix. Il recherche de nombreuses lignes de texte, généralement acheminées à partir d'une autre commande, pour une chaîne donnée : `grep`

```
use std::io;
use std::io::prelude::*;

fn grep(target: &str) -> io::Result<()> {
    let stdin = io::stdin();
    for line_result in stdin.lock().lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

Puisque nous voulons appeler , nous avons besoin d'une source d'entrée qui implémente . Dans ce cas, nous appelons pour obtenir les données qui nous sont acheminées. Cependant, la bibliothèque standard Rust protège avec un mutex. Nous appelons à verrouiller pour l'usage exclusif du fil actuel; il renvoie une valeur qui implémente . À la fin de la boucle, le est lâché, libérant le mutex. (Sans mutex, deux threads essayant de lire en même temps provoqueraient un comportement indéfini. C a le même problème et le résout de la même manière: toutes les fonctions d'entrée et de sortie standard C obtiennent un verrou dans les coulisses. La seule différence est que dans Rust, le verrou fait partie de l'API.) `.lines()` `BufRead` `io::stdin()` `stdin` `.lock()` `stdin` `Stdin` `Lock` `BufRead` `StdinLock` `stdin`

Le reste de la fonction est simple : elle appelle et boucle sur l'itérateur résultant. Étant donné que cet itérateur produit des valeurs, nous utilisons

l'opérateur pour vérifier les erreurs. `.lines() Result ?`

Supposons que nous voulions pousser notre programme un peu plus loin et ajouter la prise en charge de la recherche de fichiers sur le disque. Nous pouvons rendre cette fonction générique: `grep`

```
fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}
```

Maintenant, nous pouvons le passer soit un ou un tamponné: `StdinLock File`

```
let stdin = io::stdin();
grep(&target, stdin.lock())?;  // ok

let f = File::open(file)?;
grep(&target, BufReader::new(f))?;  // also ok
```

Notez que a n'est pas automatiquement mis en mémoire tampon. implémente mais pas . Cependant, il est facile de créer un lecteur tamponné pour un , ou tout autre lecteur sans tampon. fait cela. (Pour définir la taille de la mémoire tampon, utilisez .) `File File Read BufRead File BufReader::new(reader) BufReader::with_capacity(size, reader)`

Dans la plupart des langues, les fichiers sont mis en mémoire tampon par défaut. Si vous voulez une entrée ou une sortie sans tampon, vous devez trouver comment désactiver la mise en mémoire tampon. Dans Rust, et sont deux fonctionnalités de bibliothèque distinctes, parce que parfois vous voulez des fichiers sans mise en mémoire tampon, et parfois vous voulez mettre en mémoire tampon sans fichiers (par exemple, vous pouvez vouloir mettre en mémoire tampon l'entrée du réseau). `File BufReader`

Le programme complet, y compris la gestion des erreurs et l'analyse des arguments bruts, est illustré ici :

```
// grep - Search stdin or some files for lines matching a given string.

use std::error::Error;
use std::io::{self, BufReader};
use std::io::prelude::*;
use std::fs::File;
use std::path::PathBuf;

fn grep<R>(target: &str, reader: R) -> io::Result<()>
    where R: BufRead
{
    for line_result in reader.lines() {
        let line = line_result?;
        if line.contains(target) {
            println!("{}", line);
        }
    }
    Ok(())
}

fn grep_main() -> Result<(), Box<dyn Error>> {
    // Get the command-line arguments. The first argument is the
    // string to search for; the rest are filenames.
    let mut args = std::env::args().skip(1);
    let target = match args.next() {
        Some(s) => s,
        None => Err("usage: grep PATTERN FILE...")?
    };
    let files: Vec<PathBuf> = args.map(PathBuf::from).collect();

    if files.is_empty() {
        let stdin = io::stdin();
        grep(&target, stdin.lock())?;
    } else {
        for file in files {
            let f = File::open(file)?;
            grep(&target, BufReader::new(f))?;
        }
    }

    Ok(())
}
```

```
fn main() {
    let result = grep_main();
    if let Err(err) = result {
        eprintln!("{}", err);
        std::process::exit(1);
    }
}
```

Collecte de lignes

Plusieurs méthodes de lecture, y compris , renvoient des itérateurs qui produisent des valeurs. La première fois que vous souhaitez rassembler toutes les lignes d'un fichier en un seul grand vecteur, vous rencontrerez un problème pour vous débarrasser du s: `.lines() Result Result`

```
// ok, but not what you want
let results: Vec<io::Result<String>> = reader.lines().collect();

// error: can't convert collection of Results to Vec<String>
let lines: Vec<String> = reader.lines().collect();
```

Le deuxième essai ne compile pas : qu'adviendrait-il des erreurs ? La solution simple consiste à écrire une boucle et à vérifier chaque élément pour les erreurs: `for`

```
let mut lines = vec![];
for line_result in reader.lines() {
    lines.push(line_result?);
}
```

Pas mal; mais ce serait bien de l'utiliser ici, et il s'avère que nous le pouvons. Il suffit de savoir quel type demander : `.collect()`

```
let lines = reader.lines().collect::<io::Result<Vec<String>>>()?;
```

Comment cela fonctionne-t-il ? La bibliothèque standard contient une implémentation de pour (facile à négliger dans la documentation en ligne - qui rend cela possible : `FromIterator Result`

```
impl<T, E, C> FromIterator<Result<T, E>> for Result<C, E>
    where C: FromIterator<T>
{
```

```
    ...
}
```

Cela nécessite une lecture attentive, mais c'est une belle astuce. Supposons qu'il s'agisse de n'importe quel type de collection, comme ou . Tant que nous savons déjà comment construire un à partir d'un itérateur de valeurs, nous pouvons construire un à partir d'un itérateur produisant des valeurs. Nous avons juste besoin de tirer des valeurs de l'itérateur et de construire la collection à partir des résultats, mais si jamais nous voyons un , arrêtez-vous et transmettez-le. `C` `Vec` `HashSet` `C` `T` `Result<C, E>` `Result<T, E>` `Ok` `Err`

En d'autres termes, est un type de collection, de sorte que la méthode peut créer et remplir des valeurs de ce type. `io::Result<Vec<String>>` `.collect()`

Écrivains

Comme nous l'avons vu, la saisie se fait principalement à l'aide de méthodes. La sortie est un peu différente.

Tout au long du livre, nous avons utilisé pour produire une sortie en texte brut: `println!()`

```
println!("Hello, world!");

println!("The greatest common divisor of {:?} is {}",
         numbers, d);

println!();  // print a blank line
```

Il y a aussi une macro, qui n'ajoute pas de caractère de nouvelle ligne à la fin, et des macros qui écrivent dans le flux d'erreur standard. Les codes de mise en forme pour tous ces éléments sont les mêmes que ceux de la macro, décrits dans « Valeurs de mise en forme ». `print!` `() eprintln! eprint! format!`

Pour envoyer la sortie à un enregistreur, utilisez les macros et. Ils sont les mêmes que et , à l'exception de deux différences : `write!()` `writeln!` `()` `print!()` `println!()`

```
writeln!(io::stderr(), "error: world not helloable")?;
```

```
    writeln!(&mut byte_vec, "The greatest common divisor of {:?} is {}",
             numbers, d)?;
```

Une différence est que les macros prennent chacune un premier argument supplémentaire, un écrivain. L'autre est qu'ils renvoient un , donc les erreurs doivent être gérées. C'est pourquoi nous avons utilisé l'opérateur à la fin de chaque ligne. `write Result ?`

Les macros ne renvoient pas de ; ils paniquent simplement si l'écriture échoue. Comme ils écrivent sur le terminal, c'est rare. `print Result`

Le trait a ces méthodes: `Write`

`writer.write(&buf)`

Écrit une partie des octets de la tranche dans le flux sous-jacent. Il renvoie un fichier . En cas de succès, cela donne le nombre d'octets écrits, qui peut être inférieur à , au gré du flux. `buf io::Result<usize> buf.len()`

Comme , il s'agit d'une méthode de bas niveau que vous devriez éviter d'utiliser directement. `Reader::read()`

`writer.write_all(&buf)`

Écrit tous les octets de la tranche . Retourne. `buf Result<()>`

`writer.flush()`

Vide toutes les données mises en mémoire tampon dans le flux sous-jacent. Retourne. `Result<()>`

Notez que si les macros et vident automatiquement le flux stdout et stderr, les macros et ne le font pas. Vous devrez peut-être appeler manuellement lorsque vous les utilisez. `println! eprintln! print! eprint! flush()`

Comme les lecteurs, les écrivains sont fermés automatiquement lorsqu'ils sont abandonnés.

Tout comme ajoute un tampon à n'importe quel lecteur, ajoute un tampon à n'importe quel écrivain: `BufReader::new(reader) BufWriter::new(writer)`

```
    let file = File::create("tmp.txt")?;
    let writer = BufWriter::new(file);
```

Pour définir la taille de la mémoire tampon, utilisez `.BufWriter::with_capacity(size, writer)`

Lorsque a est supprimé, toutes les données mises en mémoire tampon restantes sont écrites dans l'enregistreur sous-jacent. Toutefois, si une erreur se produit pendant cette écriture, l'erreur est *ignorée*. (Étant donné que cela se produit à l'intérieur de la méthode de , il n'y a pas d'endroit utile pour signaler l'erreur.) Pour vous assurer que votre application remarque toutes les erreurs de sortie, mettez manuellement en mémoire tampon les rédacteurs avant de les supprimer. `BufWriter BufWriter .drop() .flush()`

Fichiers

Nous avons déjà vu deux façons d'ouvrir un fichier :

`File::open(filename)`

Ouvre un fichier existant pour lecture. Il renvoie un , et c'est une erreur si le fichier n'existe pas. `io::Result<File>`

`File::create(filename)`

Crée un nouveau fichier pour l'écriture. Si un fichier existe avec le nom de fichier donné, il est tronqué.

Notez que le type se trouve dans le module de système de fichiers, et non dans . `File std::fs std::io`

Lorsque ni l'un ni l'autre de ces éléments ne correspond à la facture, vous pouvez l'utiliser pour spécifier le comportement souhaité exact : `OpenOptions`

```
use std::fs::OpenOptions;

let log = OpenOptions::new()
    .append(true)  // if file exists, add to the end
    .open("server.log")?;

let file = OpenOptions::new()
    .write(true)
    .create_new(true)  // fail if file exists
    .open("new_file.txt")?;
```

Les méthodes , , , et ainsi de suite sont conçues pour être enchaînées comme ceci: chacune renvoie . Ce modèle de conception de chaînage de

méthode est assez commun pour avoir un nom dans Rust: il s'appelle un *constructeur*. est un autre exemple. Pour plus de détails sur , consultez la documentation en ligne. `.append()` `.write()` `.create_new()` `self` `std::process::Command` `OpenOptions`

Une fois qu'un a été ouvert, il se comporte comme n'importe quel autre lecteur ou écrivain. Vous pouvez ajouter un tampon si nécessaire. Le sera fermé automatiquement lorsque vous le déposerez. `File` `File`

Recherche

`File` implémente également le trait, ce qui signifie que vous pouvez sauter dans un plutôt que de lire ou d'écrire en un seul passage du début à la fin. est défini comme suit : `Seek` `File` `Seek`

```
pub trait Seek {
    fn seek(&mut self, pos: SeekFrom) -> io::Result<u64>;
}

pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64)
}
```

Grâce à l'enum, la méthode est joliment expressive : utiliser pour rembobiner au début et utiliser pour revenir en arrière de quelques octets, et ainsi de suite. `seek` `file.seek(SeekFrom::Start(0))` `file.seek(SeekFrom::Current(-8))`

La recherche dans un fichier est lente. Que vous utilisiez un disque dur ou un disque SSD, une recherche prend autant de temps que la lecture de plusieurs mégaoctets de données.

Autres types de lecteurs et d'écrivains

Jusqu'à présent, ce chapitre a utilisé comme exemple un cheval de bataille, mais il existe de nombreux autres types de lecteurs et d'écrivains utiles: `File`

`io::stdin()`

Renvoie un lecteur pour le flux d'entrée standard. Son type est . Comme cela est partagé par tous les threads, chaque lecture acquiert et libère un mutex. `io::Stdin`

`Stdin` possède une méthode qui acquiert le mutex et renvoie un , un lecteur tamponné qui maintient le mutex jusqu'à ce qu'il soit abandonné. Les opérations individuelles sur le évitent donc la surcharge mutex. Nous avons montré un exemple de code utilisant cette méthode dans « Reading Lines ». `.lock() io::StdinLock StdinLock`

Pour des raisons techniques, ne fonctionne pas. Le verrou contient une référence à la valeur, ce qui signifie que la valeur doit être stockée quelque part pour qu'elle vive assez longtemps : `io::stdin().lock() Stdin Stdin`

```
let stdin = io::stdin();
let lines = stdin.lock().lines();  // ok
```

`io::stdout()`, `io::stderr()`

Types de retour et d'écriture pour les flux de sortie standard et d'erreur standard. Ceux-ci aussi ont des mutex et des méthodes. `Stdout Stderr .lock()`

`Vec<u8>`

Implémente. L'écriture sur un étend le vecteur avec les nouvelles données. `Write Vec<u8>`

(`String` , cependant, n'implémente *pas* . Pour créer une chaîne à l'aide de , écrivez d'abord dans un , puis utilisez pour convertir le vecteur en chaîne.) `Write Write Vec<u8> String::from_utf8(vec)`

`Cursor::new(buf)`

Crée un , un lecteur mis en mémoire tampon qui lit à partir de . C'est ainsi que vous créez un lecteur qui lit à partir d'un fichier . L'argument peut être n'importe quel type qui implémente , de sorte que vous pouvez également passer un , , ou . `Cursor buf String buf AsRef<[u8]> &[u8] &str Vec<u8>`

`Cursor` sont triviaux en interne. Ils n'ont que deux champs: lui-même et un entier, le décalage dans l'endroit où la lecture suivante commencera. La position est initialement 0. `buf buf`

Les curseurs implémentent , et . Si le type de est ou , alors le également implémente . L'écriture sur un curseur remplace les octets en commençant à la position actuelle. Si vous essayez d'écrire au-delà de la fin d'un , vous obtiendrez une écriture partielle ou un fichier . Utiliser un curseur pour écrire au-delà de la fin de a est bien, cependant: il fait croître le vecteur. et ainsi mettre en œuvre les quatre traits. `Read BufRead Seek buf &mut`
`[u8] Vec<u8> Cursor Write buf &mut`
`[u8] io::Error Vec<u8> Cursor<&mut`
`[u8]> Cursor<Vec<u8>> std::io::prelude`

`std::net::TcpStream`

Représente une connexion réseau TCP. Puisque TCP permet la communication bidirectionnelle, c'est à la fois un lecteur et un écrivain.

La fonction associée au type tente de se connecter à un serveur et renvoie un fichier . `TcpStream::connect(("hostname", PORT)) io::Result<TcpStream>`

`std::process::Command`

Prend en charge la génération d'un processus enfant et la tuyauterie des données vers son entrée standard, comme suit :

```
use std::process::{Command, Stdio};

let mut child =
    Command::new("grep")
    .arg("-e")
    .arg("a.*e.*i.*o.*u")
    .stdin(Stdio::piped())
    .spawn()?;

let mut to_child = child.stdin.take().unwrap();
for word in my_words {
    writeln!(to_child, "{}", word)?;
}
drop(to_child);  // close grep's stdin, so it will exit
child.wait()?;
```

Le type de est ; ici, nous avons utilisé lors de la configuration du processus enfant, donc est définitivement rempli quand réussit. Si nous ne l'avions pas fait, ce serait

`.child.stdin Option<std::process::ChildStdin> .stdin(St dio::piped()) child.stdin .spawn() child.stdin None`

`Command` a également des méthodes similaires et , qui peuvent être utilisées pour demander des lecteurs dans et `..stdout() .stderr() child.stdout child.stderr`

Le module offre également une poignée de fonctions qui renvoient des lecteurs et des écrivains triviaux: `std::io`

`io::sink()`

C'est l'écrivain no-op. Toutes les méthodes d'écriture renvoient , mais les données sont simplement ignorées. `Ok`

`io::empty()`

C'est le lecteur no-op. La lecture réussit toujours, mais renvoie la fin de l'entrée.

`io::repeat(byte)`

Renvoie un lecteur qui répète l'octet donné à l'infini.

Données binaires, compression et sérialisation

De nombreuses caisses open source s'appuient sur le framework pour offrir des fonctionnalités supplémentaires. `std::io`

La caisse offre et des traits qui ajoutent des méthodes à tous les lecteurs et écrivains pour l'entrée et la sortie binaires: `byteorder ReadBytesExt WriteBytesExt`

```
use byteorder::{ReadBytesExt, WriteBytesExt, LittleEndian};

let n = reader.read_u32::<LittleEndian>()?;
writer.write_i64::<LittleEndian>(n as i64)?;
```

La caisse fournit des méthodes d'adaptateur pour la lecture et l'écriture de données ped : `flate2 gzip`

```
use flate2::read::GzDecoder;
let file = File::open("access.log.gz")?;
let mut gzip_reader = GzDecoder::new(file);
```

La caisse, et ses caisses de format associées telles que , implémentent la sérialisation et la désérialisation: elles convertissent entre les structures Rust et les octets. Nous l'avons déjà mentionné une fois, dans « Traits et

types d'autres personnes ». Maintenant, nous pouvons regarder de plus près. `serde` `serde_json`

Supposons que nous ayons des données – la carte d'un jeu d'aventure textuel – stockées dans un : `HashMap`

```
type RoomId = String;                          // each room has a unique r
type RoomExits = Vec<(char, RoomId)>;          // ...and a list of exits
type RoomMap = HashMap<RoomId, RoomExits>;     // room names and exits, si

// Create a simple map.
let mut map = RoomMap::new();
map.insert("Cobble Crawl".to_string(),
           vec![('W', "Debris Room".to_string())]);
map.insert("Debris Room".to_string(),
           vec![('E', "Cobble Crawl".to_string()),
                ('W', "Sloping Canyon".to_string())]);
...
```

La transformation de ces données en JSON pour la sortie est une seule ligne de code :

```
serde_json::to_writer(&mut std::io::stdout(), &map)?;
```

En interne, utilise la méthode du trait. La bibliothèque attache ce trait à tous les types qu'elle sait sérialiser, et cela inclut tous les types qui apparaissent dans nos données : chaînes, caractères, tuples, vecteurs et s. `serde_json::to_writer` `serialize` `serde::Serialize` `HashMap`

`serde` est flexible. Dans ce programme, la sortie est des données JSON, car nous avons choisi le sérialiseur. D'autres formats, comme MessagePack, sont également disponibles. De même, vous pouvez envoyer cette sortie à un fichier, à un ou à tout autre scripteur. Le code précédent imprime les données sur . Le voilà: `serde_json` `Vec<u8>` `stdout`

```
{"Debris Room":[["E","Cobble Crawl"],["W","Sloping Canyon"]],"Cobble Cr
[["W","Debris Room"]]}
```

`serde` inclut également la prise en charge de la dérivation des deux traits clés : `serde`

```
#[derive(Serialize, Deserialize)]
struct Player {
```

```
    location: String,
    items: Vec<String>,
    health: u32
}
```

Cet attribut peut rendre vos compilations un peu plus longues, vous devez donc demander explicitement à le prendre en charge lorsque vous le répertoriez comme dépendance dans votre fichier *Cargo.toml*. Voici ce que nous avons utilisé pour le code précédent : `#[derive] serde`

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

Consultez la documentation pour plus de détails. En bref, le système de build génère automatiquement des implémentations de et pour , de sorte que la sérialisation d'une valeur est simple
: `serde serde::Serialize serde::Deserialize Player Player`

```
serde_json::to_writer(&mut std::io::stdout(), &player)?;
```

La sortie ressemble à ceci :

```
{"location":"Cobble Crawl","items":["a wand"],"health":3}
```

Fichiers et répertoires

Maintenant que nous avons montré comment travailler avec les lecteurs et les écrivains, les quelques sections suivantes couvrent les fonctionnalités de Rust pour travailler avec des fichiers et des répertoires, qui vivent dans les modules et. Toutes ces fonctionnalités impliquent de travailler avec des noms de fichiers, nous allons donc commencer par les types de noms de fichiers. `std::path std::fs`

OsStr et chemin d'accès

Malheureusement, votre système d'exploitation ne force pas les noms de fichiers à être des Unicode valides. Voici deux commandes shell Linux qui créent des fichiers texte. Seul le premier utilise un nom de fichier UTF-8 valide :

```
$ echo "hello world" > ô.txt
$ echo "O brave new world, that has such filenames in't" > $'\xf4'.txt
```

Les deux commandes passent sans commentaire, car le noyau Linux ne connaît pas UTF-8 d'Ogg Vorbis. Pour le noyau, toute chaîne d'octets (à l'exclusion des octets nuls et des barres obliques) est un nom de fichier acceptable. C'est une histoire similaire sur Windows: presque n'importe quelle chaîne de « caractères larges » 16 bits est un nom de fichier acceptable, même les chaînes qui ne sont pas valides UTF-16. Il en va de même pour les autres chaînes gérées par le système d'exploitation, telles que les arguments de ligne de commande et les variables d'environnement.

Les chaînes Rust sont toujours des Unicode valides. Les noms de fichiers sont *presque* toujours Unicode dans la pratique, mais Rust doit faire face d'une manière ou d'une autre aux rares cas où ils ne le sont pas. C'est pourquoi Rust a et . `std::ffi::OsStr` `OsString`

`OsStr` est un type de chaîne qui est un sur-ensemble d'UTF-8. Son travail consiste à pouvoir représenter tous les noms de fichiers, les arguments de ligne de commande et les variables d'environnement sur le système actuel, *qu'ils soient Unicode valides ou non.* Sous Unix, an peut contenir n'importe quelle séquence d'octets. Sous Windows, an est stocké à l'aide d'une extension UTF-8 qui peut encoder n'importe quelle séquence de valeurs 16 bits, y compris des substituts inégalés. `OsStr` `OsStr`

Nous avons donc deux types de chaînes: pour les chaînes Unicode réelles; et pour toutes les absurdités que votre système d'exploitation peut faire. Nous en présenterons un de plus : , pour les noms de fichiers. Celui-ci est purement une commodité. est exactement comme , mais il ajoute de nombreuses méthodes pratiques liées au nom de fichier, que nous couvrirons dans la section suivante. À utiliser pour les chemins absolus et relatifs. Pour un composant individuel d'un chemin d'accès, utilisez . `str` `OsStr` `std::path::Path` `Path` `OsStr` `Path` `OsStr`

Enfin, pour chaque type de chaîne, il existe un type *de propriété* correspondant : un possède un tas alloué, un possède un tas alloué et un possède un tas alloué. Le tableau 18-1 présente certaines des caractéristiques de chaque type. `String` `str` `std::ffi::OsString` `OsStr` `std::path::PathBuf` `Path`

Tableau 18-1. Types de noms de fichiers

	Str	OsStr	Chemin
Type non dimensionné, toujours transmis par référence	Oui	Oui	Oui
Peut contenir n'importe quel texte Unicode	Oui	Oui	Oui
Ressemble à UTF-8, normalement	Oui	Oui	Oui
Peut contenir des données non Unicode	Non	Oui	Oui
Méthodes de traitement de texte	Oui	Non	Non
Méthodes liées au nom de fichier	Non	Non	Oui
Équivalent possédé, cultivable, alloué en tas	`String`	`OsString`	`PathBuf`
Convertir en type possédé	`.to_string()`	`.to_os_string()`	`.to_path_buf()`

Ces trois types implémentent un trait commun, de sorte que nous pouvons facilement déclarer une fonction générique qui accepte « n'importe quel type de nom de fichier » comme argument. Cela utilise une technique que nous avons montrée dans « AsRef et AsMut »: `AsRef<Path>`

```
use std::path::Path;
use std::io;

fn swizzle_file<P>(path_arg: P) -> io::Result<()>
    where P: AsRef<Path>
{
    let path = path_arg.as_ref();
```

```
    ...
}
```

Toutes les fonctions et méthodes standard qui prennent des arguments utilisent cette technique, de sorte que vous pouvez librement passer des littéraux de chaîne à n'importe lequel d'entre eux. `path`

Méthodes Path et PathBuf

`Path` offre les méthodes suivantes, entre autres:

`Path::new(str)`

Convertit un ou en un fichier . Cela ne copie pas la chaîne. Le nouveau pointe vers les mêmes octets que l'original ou : `&str &OsStr &Path &Path &str &OsStr`

```
use std::path::Path;
let home_dir = Path::new("/home/fwolfe");
```

(La méthode similaire convertit a en un .) `OsStr::new(str) &str &OsStr`

`path.parent()`

Renvoie le répertoire parent du chemin d'accès, le cas échéant. Le type de retour est . `Option<&Path>`

Cela ne copie pas le chemin d'accès. Le répertoire parent de est toujours une sous-chaîne de : `path path`

```
assert_eq!(Path::new("/home/fwolfe/program.txt").parent(),
           Some(Path::new("/home/fwolfe")));
```

`path.file_name()`

Renvoie le dernier composant de , le cas échéant. Le type de retour est . `path Option<&OsStr>`

Dans le cas typique, où se compose d'un répertoire, puis d'une barre oblique, puis d'un nom de fichier, cela renvoie le nom de fichier : `path`

```
use std::ffi::OsStr;
assert_eq!(Path::new("/home/fwolfe/program.txt").file_name(),
           Some(OsStr::new("program.txt")));
```

path.is_absolute(), path.is_relative()

Ceux-ci indiquent si le fichier est absolu, comme le chemin Unix */usr/bin/advent* ou le chemin Windows *C:\Program Files*, ou relatif, comme *src/main.rs*.

path1.join(path2)

Joint deux chemins, en renvoyant un nouveau : `PathBuf`

```
let path1 = Path::new("/usr/share/dict");
assert_eq!(path1.join("words"),
           Path::new("/usr/share/dict/words"));
```

Si est un chemin absolu, cela renvoie simplement une copie de , de sorte que cette méthode peut être utilisée pour convertir n'importe quel chemin en chemin absolu : `path2 path2`

```
let abs_path = std::env::current_dir()?.join(any_path);
```

path.components()

Renvoie un itérateur sur les composants du chemin d'accès donné, de gauche à droite. Le type d'élément de cet itérateur est , un enum qui peut représenter tous les différents éléments pouvant apparaître dans les noms de fichiers : `std::path::Component`

```
pub enum Component<'a> {
    Prefix(PrefixComponent<'a>),  // a drive letter or share (on Wi
    RootDir,                      // the root directory, `/` or `\`
    CurDir,                       // the `.` special directory
    ParentDir,                    // the `..` special directory
    Normal(&'a OsStr)             // plain file and directory names
}
```

Par exemple, le chemin Windows *||venice|Music|A Love Supreme|04-Psalm.mp3* se compose d'un *||venice|Music* représentant un , puis de deux composants représentant *A Love Supreme* et *04-Psalm.mp3*. `Prefix RootDir Normal`

Pour plus de détails, consultez la documentation en ligne.

path.ancestors()

Renvoie un itérateur qui se déplace jusqu'à la racine. Chaque article produit est un : d'abord lui-même, puis son parent, puis son grand-parent, et ainsi de suite : `path Path path`

```
let file = Path::new("/home/jimb/calendars/calendar-18x18.pdf");
assert_eq!(file.ancestors().collect::<Vec<_>>(),
           vec![Path::new("/home/jimb/calendars/calendar-18x18.pdf"
                Path::new("/home/jimb/calendars"),
                Path::new("/home/jimb"),
                Path::new("/home"),
                Path::new("/")]);
```

C’est un peu comme appeler à plusieurs reprises jusqu’à ce qu’il revienne . L’élément final est toujours un chemin racine ou préfixe. `parent None`

Ces méthodes fonctionnent sur des chaînes en mémoire. ont également des méthodes qui interrogent le système de fichiers : , , , , , , et ainsi de suite. Consultez la documentation en ligne pour en savoir plus. `Path .exists() .is_file() .is_dir() .read_dir() .canonicalize()`

Il existe trois méthodes pour convertir s en chaînes. Chacun d’eux permet la possibilité d’utf-8 non valide dans le : `Path Path`

`path.to_str()`

Convertit a en chaîne, sous la forme d’un fichier . Si UTF-8 n’est pas valide, cela renvoie : `Path Option<&str> path None`

```
if let Some(file_str) = path.to_str() {
    println!("{}", file_str);
}  // ...otherwise skip this weirdly named file
```

`path.to_string_lossy()`

C’est fondamentalement la même chose, mais il parvient à renvoyer une sorte de chaîne dans tous les cas. Si UTF-8 n’est pas valide, ces méthodes effectuent une copie, en remplaçant chaque séquence d’octets non valide par le caractère de remplacement Unicode, U+FFFD (' '). `path`

Le type de retour est : une chaîne empruntée ou possédée. Pour obtenir un à partir de cette valeur, utilisez sa méthode. (Pour en savoir plus sur , voir « Emprunter et posséder au travail : l’humble vache ».) `std::borrow::Cow<str> String .to_owned() Cow`

`path.display()`

Ceci est pour les chemins d'impression:

```
println!("Download found. You put it in: {}", dir_path.display());
```

La valeur renvoyée n'est pas une chaîne, mais elle implémente , de sorte qu'elle peut être utilisée avec , et des amis. Si le chemin d'accès n'est pas valide UTF-8, la sortie peut contenir le caractère. `std::fmt::Display format!() println!()`

Fonctions d'accès au système de fichiers

Le tableau 18-2 montre certaines des fonctions et leurs équivalents approximatifs sous Unix et Windows. Toutes ces fonctions renvoient des valeurs. Sauf indication contraire, ils le sont. `std::fs io::Result Result<()>`

Tableau 18-2. Résumé des fonctions d'accès au système de fichiers

	Fonction rouille	Unix	Windows
Création et suppression	`create_dir(path)`	`mkdir()`	`CreateDirectory()`
	`create_dir_all(path)`	comme `mkdir -p`	comme `mkdir`
	`remove_dir(path)`	`rmdir()`	`RemoveDirectory()`
	`remove_dir_all(path)`	comme `rm -r`	comme `rmdir /s`
	`remove_file(path)`	`unlink()`	`DeleteFile()`
Copier, déplacer et lier	`copy(src_path, dest_path) -> Result<u64>`	comme `cp -p`	`CopyFileEx()`
	`rename(src_path, dest_path)`	`rename()`	`MoveFileEx()`
	`hard_link(src_path, dest_path)`	`link()`	`CreateHardLink()`
Inspection	`canonicalize(path) -> Result<PathBuf>`	`realpath()`	`GetFinalPathNameByHandle()`
	`metadata(path) -> Result<Metadata>`	`stat()`	`GetFileInformationByHandle()`
	`symlink_metadata(path) -> Result<Metadata>`	`lstat()`	`GetFileInformationByHandle()`

Fonction rouille	Unix	Windows
`read_dir(path) -> Result<ReadDir>`	`opendir()`	`FindFirstFile()`
`read_link(path) -> Result<PathBuf>`	`readlink()`	`FSCTL_GET_REPARSE_POINT`
`set_permissions(path, perm)`	`chmod()`	`SetFileAttributes()`

Autorisations

(Le nombre renvoyé par est la taille du fichier copié, en octets. Pour créer des liens symboliques, voir « Fonctionnalités spécifiques à la plate-forme ».) `copy()`

Comme vous pouvez le voir, Rust s'efforce de fournir des fonctions portables qui fonctionnent de manière prévisible sur Windows ainsi que sur macOS, Linux et d'autres systèmes Unix.

Un tutoriel complet sur les systèmes de fichiers dépasse le cadre de ce livre, mais si vous êtes curieux de connaître l'une de ces fonctions, vous pouvez facilement en trouver plus à leur sujet en ligne. Nous montrerons quelques exemples dans la section suivante.

Toutes ces fonctions sont implémentées en appelant le système d'exploitation. Par exemple, n'utilise pas simplement le traitement de chaîne pour éliminer et à partir du fichier . Il résout les chemins relatifs à l'aide du répertoire de travail actuel et recherche les liens symboliques. C'est une erreur si le chemin n'existe pas. `std::fs::canonicalize(path) . .. path`

Type produit par et contenant des informations telles que le type et la taille du fichier, les autorisations et les horodatages. Comme toujours, consultez la documentation pour plus de détails. `Metadata std::fs::metadata(path) std::fs::symlink_metadata(path)`

Pour plus de commodité, le type a quelques-uns d'entre eux intégrés comme méthodes: , par exemple, est la même chose que . `Path path.metadata() std::fs::metadata(path)`

Lecture de répertoires

Pour lister le contenu d'un répertoire, utilisez ou, de manière équivalente, la méthode d'un : `std::fs::read_dir` `.read_dir()` `Path`

```
for entry_result in path.read_dir()? {
    let entry = entry_result?;
    println!("{}", entry.file_name().to_string_lossy());
}
```

Notez les deux utilisations de dans ce code. La première ligne vérifie les erreurs d'ouverture du répertoire. La deuxième ligne vérifie les erreurs de lecture de l'entrée suivante. `?`

Le type de est , et c'est une structure avec seulement quelques méthodes: `entry` `std::fs::DirEntry`

`entry.file_name()`

Nom du fichier ou du répertoire, sous la forme d'un fichier . `OsString`

`entry.path()`

C'est la même chose, mais avec le chemin d'origine qui y est joint, produisant un nouveau . Si le répertoire que nous répertorions est , et est , alors renvoie . `PathBuf` `"/home/jimb"` `entry.file_name()` `".emacs"` `entry.path()` `PathBuf::from("/home/jimb/.emacs")`

`entry.file_type()`

Renvoie un fichier . a , et méthodes. `io::Result<FileType>` `FileType` `.is_file()` `.is_dir()` `.is_symlink()`

`entry.metadata()`

Obtient le reste des métadonnées relatives à cette entrée.

Les répertoires spéciaux et *ne sont pas* répertoriés lors de la lecture d'un répertoire. `.` `..`

Voici un exemple plus substantiel. Le code suivant copie récursivement une arborescence de répertoires d'un emplacement à un autre sur le disque :

```
use std::fs;
use std::io;
use std::path::Path;

/// Copy the existing directory `src` to the target path `dst`.
fn copy_dir_to(src: &Path, dst: &Path) -> io::Result<()> {
```

```
        if !dst.is_dir() {
            fs::create_dir(dst)?;
        }

        for entry_result in src.read_dir()? {
            let entry = entry_result?;
            let file_type = entry.file_type()?;
            copy_to(&entry.path(), &file_type, &dst.join(entry.file_name())
        }

        Ok(())
    }
```

Une fonction distincte, , copie les entrées de répertoire individuelles
: `copy_to`

```
    /// Copy whatever is at `src` to the target path `dst`.
    fn copy_to(src: &Path, src_type: &fs::FileType, dst: &Path)
        -> io::Result<()>
    {
        if src_type.is_file() {
            fs::copy(src, dst)?;
        } else if src_type.is_dir() {
            copy_dir_to(src, dst)?;
        } else {
            return Err(io::Error::new(io::ErrorKind::Other,
                                      format!("don't know how to copy: {}",
                                              src.display())));
        }
        Ok(())
    }
```

Fonctionnalités spécifiques à la plate-forme

Jusqu'à présent, notre fonction peut copier des fichiers et des répertoires. Supposons que nous voulions également prendre en charge les liens symboliques sous Unix. `copy_to`

Il n'existe aucun moyen portable de créer des liens symboliques qui fonctionnent à la fois sous Unix et Windows, mais la bibliothèque standard offre une fonction spécifique à Unix : `symlink`

```
    use std::os::unix::fs::symlink;
```

Avec cela, notre travail est facile. Il suffit d'ajouter une branche à l'expression dans : `if copy_to`

```
...
} else if src_type.is_symlink() {
    let target = src.read_link()?;
    symlink(target, dst)?;
...
```

Cela fonctionnera tant que nous compilerons notre programme uniquement pour les systèmes Unix, tels que Linux et macOS.

Le module contient diverses fonctionnalités spécifiques à la plate-forme, telles que . Le corps réel de dans la bibliothèque standard ressemble à ceci (en prenant une licence poétique): `std::os symlink std::os`

```
//! OS-specific functionality.

#[cfg(unix)]                    pub mod unix;
#[cfg(windows)]                 pub mod windows;
#[cfg(target_os = "ios")]       pub mod ios;
#[cfg(target_os = "linux")]     pub mod linux;
#[cfg(target_os = "macos")]     pub mod macos;
...
```

L'attribut indique une compilation conditionnelle : chacun de ces modules n'existe que sur certaines plateformes. C'est pourquoi notre programme modifié, utilisant , compilera avec succès uniquement pour Unix: sur d'autres plates-formes, n'existe pas. `#[cfg] std::os::unix std::os::unix`

Si nous voulons que notre code soit compilé sur toutes les plates-formes, avec la prise en charge des liens symboliques sous Unix, nous devons également l'utiliser dans notre programme. Dans ce cas, il est plus facile d'importer sur Unix, tout en définissant notre propre stub sur d'autres systèmes : `#[cfg] symlink symlink`

```
#[cfg(unix)]
use std::os::unix::fs::symlink;

/// Stub implementation of `symlink` for platforms that don't provide i
#[cfg(not(unix))]
fn symlink<P: AsRef<Path>, Q: AsRef<Path>>(src: P, _dst: Q)
    -> std::io::Result<()>
```

```
    {
        Err(io::Error::new(io::ErrorKind::Other,
                           format!("can't copy symbolic link: {}",
                                   src.as_ref().display())))
    }
```

Il s'avère que c'est un cas particulier. La plupart des fonctionnalités spécifiques à Unix ne sont pas des fonctions autonomes, mais plutôt des caractéristiques d'extension qui ajoutent de nouvelles méthodes aux types de bibliothèque standard. (Nous avons couvert les traits d'extension dans « Traits et types d'autres personnes ».) Il existe un module qui peut être utilisé pour activer toutes ces extensions à la fois : `symlink prelude`

```
use std::os::unix::prelude::*;
```

Par exemple, sous Unix, cela ajoute une méthode à , fournissant l'accès à la valeur sous-jacente qui représente les autorisations sous Unix. De même, il s'étend avec des accesseurs pour les champs de la valeur sous-jacente, tels que , l'ID utilisateur du propriétaire du fichier. `.mode() std::fs::Permissions u32 std::fs::Metadata st ruct stat .uid()`

Tout compte fait, ce qu'il y a dedans est assez basique. Beaucoup plus de fonctionnalités spécifiques à la plate-forme sont disponibles via des caisses tierces, comme `winreg` pour accéder au registre Windows. `std::os`

Réseautage

Un tutoriel sur le réseautage dépasse largement le cadre de ce livre. Cependant, si vous en savez déjà un peu plus sur la programmation réseau, cette section vous aidera à démarrer avec la mise en réseau dans Rust.

Pour le code réseau de bas niveau, commencez par le module, qui fournit une prise en charge multiplateforme pour la mise en réseau TCP et UDP. Utilisez la caisse pour la prise en charge SSL/TLS. `std::net native_tls`

Ces modules fournissent les blocs de construction pour une entrée et une sortie simples et bloquantes sur le réseau. Vous pouvez écrire un serveur simple en quelques lignes de code, en utilisant et en générant un thread pour chaque connexion. Par exemple, voici un serveur « echo » : `std::net`

```
use std::net::TcpListener;
use std::io;
use std::thread::spawn;

/// Accept connections forever, spawning a thread for each one.
fn echo_main(addr: &str) -> io::Result<()> {
    let listener = TcpListener::bind(addr)?;
    println!("listening on {}", addr);
    loop {
        // Wait for a client to connect.
        let (mut stream, addr) = listener.accept()?;
        println!("connection received from {}", addr);

        // Spawn a thread to handle this client.
        let mut write_stream = stream.try_clone()?;
        spawn(move || {
            // Echo everything we receive from `stream` back to it.
            io::copy(&mut stream, &mut write_stream)
                .expect("error in client thread: ");
            println!("connection closed");
        });
    }
}

fn main() {
    echo_main("127.0.0.1:17007").expect("error: ");
}
```

Un serveur d'écho répète simplement tout ce que vous lui envoyez. Ce type de code n'est pas si différent de ce que vous écririez en Java ou en Python. (Nous couvrirons dans le chapitre suivant.) `std::thread::spawn()`

Toutefois, pour les serveurs hautes performances, vous devrez utiliser des entrées et des sorties asynchrones. Le chapitre 20 couvre la prise en charge de rust pour la programmation asynchrone et montre le code complet d'un client et d'un serveur réseau.

Les protocoles de niveau supérieur sont pris en charge par des caisses tierces. Par exemple, la caisse offre une belle API pour les clients HTTP. Voici un programme complet de ligne de commande qui récupère n'importe quel document avec une URL ou un vidage sur votre terminal. Ce code a été écrit à l'aide de , avec sa fonctionnalité activée. fournit égale-

ment une interface asynchrone. `reqwest http: https: reqwest = "0.11" "blocking" reqwest`

```
use std::error::Error;
use std::io;

fn http_get_main(url: &str) -> Result<(), Box<dyn Error>> {
    // Send the HTTP request and get a response.
    let mut response = reqwest::blocking::get(url)?;
    if !response.status().is_success() {
        Err(format!("{}", response.status()))?;
    }

    // Read the response body and write it to stdout.
    let stdout = io::stdout();
    io::copy(&mut response, &mut stdout.lock())?;

    Ok(())
}

fn main() {
    let args: Vec<String> = std::env::args().collect();
    if args.len() != 2 {
        eprintln!("usage: http-get URL");
        return;
    }

    if let Err(err) = http_get_main(&args[1]) {
        eprintln!("error: {}", err);
    }
}
```

Le framework pour les serveurs HTTP offre des touches de haut niveau telles que les et traits, qui vous aident à composer une application à partir de parties enfichables. La caisse implémente le protocole WebSocket. Et ainsi de suite. Rust est un langage jeune avec un écosystème open source occupé. La prise en charge de la mise en réseau se développe rapidement. `actix-web Service Transform websocket`