

Chapitre 18. Entrée et sortie

Doolittle : Quelles preuves concrètes avez-vous de votre existence ?

Bombe #20 : Hmmm... eh bien... Je pense, donc je 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

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

- Les valeurs qui implémentent `Read` ont des méthodes pour une entrée orientée octet. Ils s'appellent *des lecteurs*.
- Les valeurs qui implémentent `BufRead` sont des lecteurs *tamponnés*. Ils prennent en charge toutes les méthodes de `Read`, ainsi que les méthodes de lecture de lignes de texte, etc.
- Des valeurs qui mettent en œuvre `Write` le support sortie de texte orientée octet et UTF-8. On les appelle *des écrivains*.

[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 lecteurs et d'écrivains présentés dans la figure et montrerons d'autres façons d'interagir avec les fichiers, le terminal et le réseau.

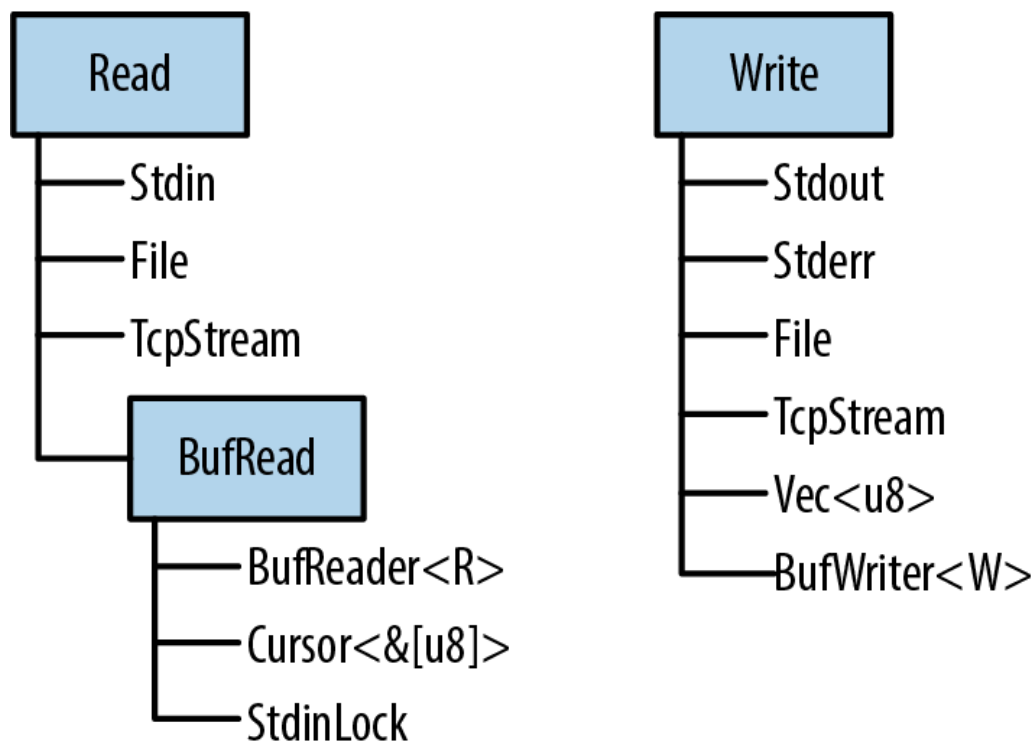


Image 18-1. Les trois principaux traits d'E/S de Rust et les types sélectionnés qui les implémentent

Lecteurs et écrivains

Les lecteurs sont des valeurs que votre programme peut lire des octets à partir de. Les exemples comprennent:

- 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 lire à partir du flux d'entrée standard du processus
- `std::io::Cursor<&[u8]>` et `std::io::Cursor<Vec<u8>>` les valeurs, qui sont des lecteurs qui "lisent" à partir d'un tableau d'octets ou d'un vecteur déjà en mémoire

Écrivains sont des valeurs sur lesquelles votre programme peut écrire des octets. Les exemples comprennent:

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

tampon, car il ne s'agit que d'une tranche d'un tableau d'octets existant

Puisqu'il existe des traits standard pour les lecteurs et les rédacteurs (`std::io::Read` et `std::io::Write`), 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 :

```
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 => continue,
            Err(e) => return Err(e),
        };
        writer.write_all(&buf[..len])?;
        written += len as u64;
    }
}
```

Il s'agit de la mise en œuvre `std::io::copy()` de Bibliothèque standard de Rust. Puisqu'il est générique, vous pouvez l'utiliser pour copier des données d'un `File` vers un `TcpStream`, d' `Stdin` un vers un en mémoire `Vec<u8>`, etc.

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

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

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

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

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

Le mot- `self` clé déclare ici `io` comme alias du `std::io` module. De cette façon, `std::io::Result` et `std::io::Error` peuvent être écrits de manière plus concise comme `io::Result` et `io::Error`, et ainsi de suite.

Lecteurs

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

```
reader.read(&mut buffer)
```

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

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

En cas d'erreur, `.read()` renvoie `Err(err)`, où `err` est une `io::Error` valeur. An `io::Error` est imprimable, pour le bénéfice des humains ; pour les programmes, il a une `.kind()` méthode qui renvoie un code d'erreur de type `io::ErrorKind`. Les membres de cette énumération ont des noms comme `PermissionDenied` et `ConnectionReset`. La plupart indiquent des erreurs graves qui ne peuvent être ignorées, mais un type d'erreur doit être traité spécialement.

`io::ErrorKind::Interrupted` correspond au code d'erreur Unix `EINTR`, 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 `copy()`, dans la section précédente, en montre un exemple.

Comme vous pouvez le voir, la `.read()` méthode est de très bas niveau, héritant même des bizarreries du système d'exploitation

sous-jacent. Si vous implémentez le `Read` trait pour un nouveau type de source de données, cela vous donne beaucoup de latitude. Si vous essayez de lire certaines données, c'est pénible. Par conséquent, Rust fournit plusieurs méthodes pratiques de niveau supérieur. Tous ont des implémentations par défaut en termes de `.read()`. Ils gèrent tous `ErrorKind::Interrupted`, vous n'avez donc pas à le faire.

```
reader.read_to_end(&mut byte_vec)
```

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

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

```
reader.read_to_string(&mut string)
```

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

Dans certains langages de programmation, la saisie d'octets et la saisie de caractères sont gérées par des types différents. 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 `encoding` caisse open source.

```
reader.read_exact(&mut buf)
```

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

Ce sont les principales méthodes du `Read` trait. De plus, il existe trois méthodes d'adaptation qui prennent le `reader` par valeur, le transformant en un itérateur ou un lecteur différent :

```
reader.bytes()
```

Retourun itérateur sur les octets du flux d'entrée. Le type d'élément est `io::Result<u8>`, donc une vérification d'erreur est requise pour chaque

octet. De plus, cela appelle `reader.read()` une fois par octet, ce qui sera très inefficace si le lecteur n'est pas mis en mémoire tampon.

```
reader.chain(reader2)
```

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

```
reader.take(n)
```

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

Il n'y a pas de méthode pour fermer un lecteur. Les lecteurs et les écrivains implémentent généralement `Drop` ainsi qu'ils se ferment automatiquement.

Lecteurs tamponnés

Pour plus d'efficacité, les lecteurs et les écrivains peuvent être mis en *mémoire tampon*, ce qui signifie simplement qu'ils ont un morceau de mémoire (un tampon) qui contient certaines 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 `BufReader`, dans cet exemple en appelant sa `.read_line()` méthode. Le `BufReader` système d'exploitation reçoit à son tour son entrée en plus gros morceaux.

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

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

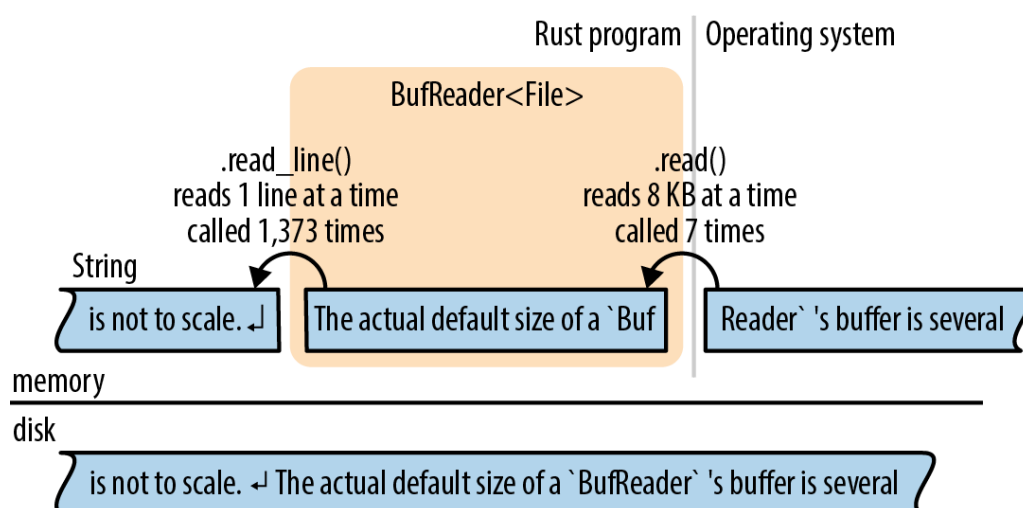


Image 18-2. Un lecteur de fichiers tamponné

Les lecteurs tamponnés implémentent à la fois `Read` et un deuxième trait, `BufRead`, qui ajoute les méthodes suivantes :

```
reader.read_line(&mut line)
```

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

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

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

```
reader.lines()
```

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

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)
```

Cessent comme `.read_line()` et `.lines()`, mais orientés octets, produisant `Vec<u8>`s au lieu de `Strings`. Vous choisissez le délimiteur `stop_byte`.

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

Les deux sections suivantes traitent plus en détail des lecteurs tamponnés.

Lignes de lecture

Voici une fonction `grep` qui implémente l'utilitaire Unix. Il recherche de nombreuses lignes de texte, généralement transmises par une autre com-

mande, pour une chaîne donnée :

```
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 `.lines()`, nous avons besoin d'une source d'entrée qui implémente `BufRead`. Dans ce cas, nous appelons `io::stdin()` pour obtenir les données qui nous sont transmises. Cependant, la bibliothèque standard Rust protège `stdin` avec un mutex. Nous appelons `.lock()` to lock `stdin` pour l'usage exclusif du thread actuel ; il renvoie une `StdinLock` valeur qui implémente `BufRead`. À la fin de la boucle, le `StdinLock` est supprimé, libérant le mutex. (Sans mutex, deux threads essayant de lire `stdin` en même temps entraîneraient 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 en arrière-plan. Le seul la différence est que dans Rust, le verrou fait partie de l'API.)

Le reste de la fonction est simple : elle appelle `.lines()` et boucle sur l'itérateur résultant. Étant donné que cet itérateur produit des `Result` valeurs, nous utilisons l' `?` opérateur pour vérifier les erreurs.

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

```
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 lui passer un `StdinLock` ou un `buffered 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 `File` n'est pas automatiquement mis en mémoire tampon. `File` implémente `Read` mais pas `BufRead`. Cependant, il est facile de créer un lecteur tamponné pour un `File`, ou tout autre lecteur non tamponné. `BufReader::new(reader)` est ce que ça. (Pour définir la taille du tampon, utilisez `BufReader::with_capacity(size, reader)`.)

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

Le programme complet, y compris la gestion des erreurs et une analyse grossière des arguments, est présenté 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);
    }
}

```

Lignes de collecte

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

```

// 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'advierait-il des erreurs ? La solution simple consiste à écrire une `for` boucle et à vérifier chaque élément pour les erreurs :

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

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

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

Comment cela marche-t-il? La bibliothèque standard contient une implémentation de `FromIterator` for `Result` — facile à oublier dans la documentation en ligne — qui rend cela possible :

```
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 bonne astuce. Supposons qu'il s'agisse `C` de n'importe quel type de collection, comme `Vec` ou `HashSet`. Du moment que l'on sait construire `a` à `C` partir d'un itérateur de `T` valeurs, on peut construire `a` à `Result<C, E>` partir d'un itérateur produisant des `Result<T, E>` valeurs. Nous avons juste besoin de tirer des valeurs de l'itérateur et de construire la collection à partir des `Ok` résultats, mais si jamais nous voyons un `Err`, arrêtez-vous et transmettez-le.

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

Écrivains

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

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

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

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

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

Il y a aussi une `print!()` macro, qui n'ajoute pas de caractère de saut de ligne à la fin, et `eprintln!` et `eprint!` les macros qui écrivent dans le flux d'erreurs standard. Les codes de formatage pour tous ces éléments sont les mêmes que ceux de la `format!` macro, décrits dans ["Formatage des valeurs"](#).

Pour envoyer la sortie à un rédacteur, utilisez les macros `write!` `()` et `writeln!()`. Ils sont identiques à `print!()` et `println!()`, à deux différences près :

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

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

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

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

Le `Write` trait a ces méthodes:

```
writer.write(&buf)
```

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

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

```
writer.write_all(&buf)
```

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

```
writer.flush()
```

Libère toutes les données mises en mémoire tampon au flux sous-jacent. Retourne `Result<()>`.

Notez que bien que les macros `println!` et `eprintln!` vident automatiquement le flux `stdout` et `stderr`, les macros `print!` et `ne le font pas`. `eprint!` Vous devrez peut-être appeler `flush()` manuellement lors de leur utilisation.

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

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

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

Pour définir la taille du tampon, utilisez

```
BufWriter::with_capacity(size, writer).
```

Lorsqu'un `BufWriter` est supprimé, toutes les données restantes en mémoire tampon sont écrites dans l'enregistreur sous-jacent. Cependant, si une erreur survient lors de cette écriture, l'erreur est *ignorée*. (Comme cela se produit dans `BufWriter` la `.drop()` 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 `.flush()` les écrivains en mémoire tampon avant de les supprimer.

Des dossiers

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

```
File::open(filename)
```

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

```
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 `File` type est dans le module de système de fichiers, `std::fs`, pas `std::io`.

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

```
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 `.append()`, `.write()`, `.create_new()`, etc. sont conçues pour être chaînées comme ceci : chacune renvoie `self`. Ce modèle de conception de chaînage de méthodes est suffisamment courant pour avoir un nom dans Rust : il s'appelle un *builder*.

`std::process::Command` est un autre exemple. Pour plus de détails sur `OpenOptions`, consultez la documentation en ligne.

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

En cherchant

`File` met également en œuvre le `Seek` trait, ce qui signifie que vous pouvez sauter dans un `File` plutôt que de lire ou d'écrire en une seule passe du début à la fin. `Seek` est défini comme ceci :

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

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

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

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'enregistreurs

Jusqu'à présent, ce chapitre a utilisé `File` comme exemple le bourreau de travail, mais il existe de nombreux autres types de lecteurs et de rédacteurs utiles :

`io::stdin()`

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

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

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

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

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

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

`Vec<u8>`

Met en œuvre `write`. L'écriture à a `Vec<u8>` étend le vecteur avec les nouvelles données.

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

`Cursor::new(buf)`

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

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

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

`std::net::TcpStream`

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

La fonction associée au type

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

`std::process::Command`

Prend en charge la création d'un processus enfant et diriger les données vers son entrée standard, comme ceci :

```
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 `typed` `child.stdin` est

`Option<std::process::ChildStdin>`; ici, nous avons utilisé `.stdin(Stdio::piped())` lors de la configuration du processus enfant, il `child.stdin` est donc définitivement rempli en cas de `.spawn()` réussite. Si nous ne l'avons pas fait, `child.stdin` ce serait `None`.

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

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

```
io::sink()
```

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

```
io::empty()
```

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

```
io::repeat(byte)
```

Renvoie un lecteur qui répète indéfiniment l'octet donné.

Données binaires, compression et sérialisation

De nombreux open sourceles caisses s'appuient sur le `std::io` cadre pour offrir des fonctionnalités supplémentaires.

La `byteorder` caisseoffres `ReadBytesExt` et `WriteBytesExt` caractéristiquesqui ajoutent des méthodes à tous les lecteurs et écrivains pour le binaireentrée et sortie :

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

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

La `flate2` caisse fournit des méthodes d'adaptation pour la lecture et écrire `gzip` des données `ped` :

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

La `serde` caisse, et ses caisses de format associées tels que `serde_json`, implémenter la sérialisation et désérialisation : ils convertissent dans les deux sens entre les structures Rust et les octets. Nous l'avons mentionné une fois auparavant, dans ["Traits et types d'autres personnes"](#). Maintenant, nous pouvons regarder de plus près.

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 name
type RoomExits = Vec<(char, RoomId)>; // ...and a list of exits
type RoomMap = HashMap<RoomId, RoomExits>; // room names and exits, simplified

// 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())]);
...
```

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

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

En interne, `serde_json::to_writer` utilise la `serialize` méthode du `serde::Serialize` 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 `HashMap`s.

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

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

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

```
#[derive(Serialize, Deserialize)]
struct Player {
    location: String,
    items: Vec<String>,
    health: u32
}
```

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

```
[dépendances]
serde = { version = "1.0", fonctionnalités = ["dériver"] }
serde_json = "1.0"
```

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

```
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 des lecteurs et des rédacteurs, les prochaines sections couvrent les fonctionnalités de Rust pour travailler avec des fichiers, et, qui résident dans les modules `std::path` et `std::fs`. Toutes ces fonctionnalités impliquent de travailler avec des noms de fichiers, nous allons donc commencer par les types de noms de fichiers.

OsStr et Chemin

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

```
$ 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 l'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 toutes les chaînes de "caractères larges" 16 bits sont un nom de fichier acceptable, même les chaînes qui ne sont pas valides en UTF-16. Il en va de même pour les autres chaînes que le système d'exploitation gère, comme les arguments de ligne de commande et les variables d'environnement.

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

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

Nous avons donc deux types de chaînes : `str` pour les chaînes Unicode réelles ; et `OsStr` pour toutes les bêtises que votre système d'exploitation peut produire. Nous allons en introduire un de plus : `std::path::Path`, pour les noms de fichiers. Celui-ci est purement une commodité. `Path` est

exactement comme `OsStr` , mais il ajoute de nombreuses méthodes pratiques liées aux noms de fichiers, que nous aborderons dans la section suivante. À utiliser `Path` pour les chemins absolus et relatifs. Pour un composant individuel d'un chemin, utilisez `OsStr` .

Enfin, pour chaque type de chaîne, il existe un *propriétaire correspondant* : `a::String` possède un heap-allocated `str` , `a::std::ffi::OsString` possède un heap-allocated `OsStr` et `a::std::path::PathBuf` possède un heap-allocated `Path` . [Le tableau 18-1](#) décrit certaines des caractéristiques de chaque type.

Tableau 18-1. Types de noms de fichiers

	chaîne	OsStr	Chemin
Type non dimensionné, toujours passé 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 aux noms de fichiers	Non	Non	Oui
Équivalent détenu, extensible et alloué en tas	<code>String</code>	<code>OsString</code>	<code>PathBuf</code>
Convertir en type possédé	<code>.to_string()</code>	<code>.to_os_string()</code>	<code>.to_path_buf()</code>

Ces trois types implémentent un trait commun, `AsRef<Path>` , nous pouvons donc 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 »](#) :

```

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 `path` des arguments utilisent cette technique, vous pouvez donc librement passer des littéraux de chaîne à n'importe laquelle d'entre elles..

Méthodes Path et PathBuf

`Path` propose les méthodes suivantes, entre autres :

Path::new(str)

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

```

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

```

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

path.parent()

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

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

```

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

```

path.file_name()

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

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

```
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()`

Ces indicateurs 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)`

Jointure de deux chemins, retournant un nouveau `PathBuf` :

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

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

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

`path.components()`

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

```
pub enum Component<'a> {
    Prefix(PrefixComponent<'a>), // a drive letter or share (on Wind
    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 `Prefix` représentant `||ve-`

`nice|Music`, suivi d'un `RootDir`, puis de deux `Normal` composants représentant `A Love Supreme` et `04-Psalm.mp3`.

Pour plus de détails, consultez [la documentation en ligne](#).

`path.ancestors()`

Retourne un itérateur qui marche de `path` haut en bas jusqu'à la racine. Chaque élément produit est un `Path`: d'abord `path` lui-même, puis son parent, puis son grand-parent, et ainsi de suite :

```
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 `parent` plusieurs reprises jusqu'à ce qu'il revienne `None`. L'élément final est toujours une racine ou un chemin de préfixe.

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

Il existe trois méthodes pour convertir des `Path`s en chaînes. Chacun permet la possibilité d'un UTF-8 invalide dans `Path` :

`path.to_str()`

Convertit un `Path` à une chaîne, comme un `Option<&str>`. Si `path` n'est pas valide UTF-8, cela retourne `None` :

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

`path.to_string_lossy()`

Cette est fondamentalement la même chose, mais il parvient à renvoyer une sorte de chaîne dans tous les cas. Si `path` n'est pas UTF-8 valide, ces méthodes font une copie, en remplaçant chaque sé-


quence d'octets invalide par le caractère de remplacement Unicode, U+FFFD ('').

Le type de retour est `std::borrow::Cow<str>` : une chaîne empruntée ou détenue. Pour obtenir a à `String` partir de cette valeur, utilisez sa `.to_owned()` méthode. (Pour en savoir plus sur `Cow`, consultez « [Emprunter et posséder au travail : la vache humble](#) ».)

```
path.display()
```

Cette 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 `std::fmt::Display`, elle peut donc être utilisée avec `format!()`, `println!()` et `friends`. Si le chemin n'est pas valide UTF-8, la sortie peut contenir le caractère .

Fonctions d'accès au système de fichiers

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

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

	Fonction rouille	Unix	les fenêtres
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()
	copy(src_path, dest_path) -> Result<u64>	Comme cp -p	CopyFileEx()
	rename(src_path, dest_path)	rename()	MoveFileEx()
Copier, déplacer et lier Inspecter	hard_link(src_path, dest_path)	link()	CreateHardLink()
	canonicalize(path) -> Result<PathBuf>	realpath()	GetFinalPathNameByHandle()
	metadata(path) -> Result<Metadata>	stat()	GetFileInformationByHandle()
	symlink_metadata(path) -> Result<Metadata>	lstat()	GetFileInformationByHandle()
	read_dir(path) -> Result<ReadDir>	opendir()	FindFirstFile()

Fonction rouille	Unix	les fenêtres
<code>read_link(path) -> Result<PathBuf></code>	<code>readlink()</code>	<code>FSCTL_GET_REPARSE_POINT</code>
<code>set_permissions(path, perm)</code>	<code>chmod()</code>	<code>SetFileAttributes()</code>

Autorisations

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

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 didacticiel 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 en ligne. Nous montrerons quelques exemples dans la section suivante.

Toutes ces fonctions sont implémentées en appelant le système d'exploitation. Par exemple, `std::fs::canonicalize(path)` ne se contente pas d'utiliser le traitement des chaînes pour éliminer `.` et `..` du donné `path`. 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.

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

Pour plus de commodité, le `Path` type a quelques-unes de ces méthodes intégrées : `path.metadata()`, par exemple, est la même chose comme `std::fs::metadata(path)`.

Répertoires de lecture

Pour lister le contenu d'un répertoire, utiliser `std::fs::read_dir` ou, de manière équivalente, la `.read_dir()` méthode d'un `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 `entry` est `std::fs::DirEntry`, et c'est une structure avec quelques méthodes :

`entry.file_name()`

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

`entry.path()`

Cette est le même, mais avec le chemin d'origine qui lui est joint, produisant un nouveau fichier `PathBuf`. Si le répertoire que nous listons est `"/home/jimb"`, et `entry.file_name()` est `".emacs"`, alors `entry.path()` retournerait `PathBuf::from("/home/jimb/.emacs")`.

`entry.file_type()`

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

`entry.metadata()`

Obtient le reste des métadonnées sur cette entrée.

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

Voici un exemple plus conséquent. Le code suivant copie de manière récursive 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, `copy_to`, copie les entrées individuelles du répertoire:

```

/// 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 `copy_to` fonction peut copier des fichiers et répertoires. Supposons que nous souhaitions également prendre en charge les liens symboliques sous Unix.

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

```

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

```

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

```

...
} else if src_type.is_symlink() {

```

• • •

ment pour les systèmes Unix, tels que Linux et macOS.

bibliothèque standard ressemble à ceci (en prenant une licence poétique):

...

```
std::os::unix n'existe pas.
```

symlink stub sur d'autres systemes :

}

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

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

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

Tout compte fait, ce qu'il y a `std::os` 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.

La mise en réseau

Un tuto sur le réseautage dépasse largement le cadre de ce livre. Cependant, si vous connaissez déjà un peu 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 `std::net` module, qui fournit une prise en charge multiplateforme pour la mise en réseau TCP et UDP. Utilisez la `native_tls` caisse pour la prise en charge SSL/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 `std::net` et en créant un thread pour chaque connexion. Par exemple, voici un serveur "echo":

```
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 `std::thread::spawn()` dans [le chapitre suivant](#).)

Cependant, pour les serveurs hautes performances, vous devrez utiliser une entrée et une sortie asynchrones. [Le chapitre 20](#) couvre la prise en charge par Rust de la programmation asynchrone et montre le code complet pour un client et un serveur réseau.

Les protocoles de niveau supérieur sont pris en charge par des caisses tierces. Par exemple, la `reqwest` caisse offre une belle API pour les clients HTTP. Voici un programme complet en ligne de commande qui récupère tout document avec une URL `http:` ou `https:` et le vide sur votre terminal. Ce code a été écrit en utilisant `reqwest = "0.11"`, avec sa "blocking" fonctionnalité activée. `reqwest` fournit également une interface asynchrone.

```
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 `actix-web` cadre pour les serveurs HTTP offre des touches de haut niveau telles que les traits `Service` et `Transform`, qui vous aident à composer une application à partir de parties enfichables. La `websocket` caisse implémente le protocole WebSocket. Etc. Rust est un langage jeune avec un écosystème open source très actif. Prise en charge de la mise en réseau est en pleine expansion.

[Soutien](#) [Se déconnecter](#)