Chapitre 19. Concurrence

> *À long terme, il n'est pas conseillé d'écrire de gros programmes concurrents dans des langages orientés machine qui permettent une utilisation illimitée des emplacements des magasins et de leurs adresses. Il n'y a tout simplement aucun moyen de rendre ces programmes fiables (même avec l'aide de mécanismes matériels compliqués).*
>
> —Par Brinch Hansen (1977)

> *Les modèles de communication sont des modèles de parallélisme.*
>
> —Whit Morris

Si votre attitude envers la concurrencea changé au cours de votre carrière, vous n'êtes pas seul. C'est une histoire commune.

Au début, écrire du code concurrent est facile et amusant. Les outils (threads, verrous, files d'attente, etc.) sont faciles à comprendre et à utiliser. Il y a beaucoup d'écueils, c'est vrai, mais heureusement vous les connaissez tous et vous faites attention à ne pas vous tromper.

À un moment donné, vous devez déboguer le multithread de quelqu'un d'autrecode, et vous êtes obligé de conclure que *certaines* personnes ne devraient vraiment pas utiliser ces outils.

Ensuite, à un moment donné, vous devez déboguer votre propre code multithread.

L'expérience inculque un scepticisme sain, voire un cynisme pur et simple, envers tout code multithread. Ceci est aidé par l'article occasionnel expliquant avec des détails abrutissants pourquoi certains idiomes de multithreading manifestement corrects ne fonctionnent pas du tout. (Cela a à voir avec "le modèle de mémoire".) Mais vous finissez par trouver une approche de la concurrence que vous pensez pouvoir utiliser de manière réaliste sans faire constamment d'erreurs. Vous pouvez intégrer à peu près tout dans cet idiome, et (si vous êtes *vraiment* bon) vous apprenez à dire « non » à une complexité accrue.

Bien sûr, il y a beaucoup d'idiomes. Les approches couramment utilisées par les programmeurs système sont les suivantes :

- Un *fil de fond*qui a un seul travail et se réveille périodiquement pour le faire.
- *Pools de travailleurs* à usage généralqui communiquent avec les clients via *des files d'attente de tâches* .
- *Pipelines*où les données circulent d'un thread à l'autre, chaque thread effectuant une petite partie du travail.
- *Parallélisme des données*, où l'on suppose (à tort ou à raison) que l'ensemble de l'ordinateur ne fera principalement qu'un gros calcul, qui est donc divisé en *n* morceaux et exécuté sur *n* threads dans l'espoir de faire fonctionner les *n* cœurs de la machine en même temps .
- *Une mer de synchronisationobjects* , où plusieurs threads ont accès aux mêmes données, et les courses sont évitées en utilisant des schémas de *verrouillage* ad hoc basés sur des primitives de bas niveau comme les mutex. (Java inclut un support intégré pour ce modèle, qui était très populaire dans les années 1990 et 2000.)
- *Les opérations sur les nombres entiers atomiques* permettent à plusieurs cœurs de communiquer en transmettant des informations à travers des champs de la taille d'un mot machine. (Ceci est encore plus difficile à obtenir que tous les autres, à moins que les données échangées ne soient littéralement que des valeurs entières. En pratique, il s'agit généralement de pointeurs.)

Avec le temps, vous pourrez peut-être utiliser plusieurs de ces approches et les combiner en toute sécurité. Vous êtes un maître de l'art. Et tout irait bien si seulement personne d'autre n'était autorisé à modifier le système de quelque manière que ce soit. Les programmes qui utilisent bien les threads regorgent de règles non écrites.

Rust offre une meilleure façon d'utiliser la concurrence, non pas en forçant tous les programmes à adopter un style unique (ce qui pour les programmeurs système ne serait pas du tout une solution), mais en prenant en charge plusieurs styles en toute sécurité. Les règles non écrites sont écrites - dans le code - et appliquées par le compilateur.

Vous avez entendu dire que Rust vous permet d'écrire des programmes sûrs, rapides et simultanés. C'est le chapitre où nous vous montrons comment c'est fait. Nous allons couvrir trois façons d'utiliser les threads Rust :

- Parallélisme fourche-jointure
- Canaux
- État mutable partagé

En cours de route, vous allez utiliser tout ce que vous avez appris jusqu'à présent sur le langage Rust. Le soin que Rust prend avec les références, la

mutabilité et les durées de vie est suffisamment précieux dans les programmes à un seul thread, mais c'est dans la programmation concurrente que la véritable signification de ces règles devient apparente. Ils permettent d'élargir votre boîte à outils, de pirater rapidement et correctement plusieurs styles de code multithread - sans scepticisme, sans cynisme, sans peur.

Parallélisme fork-join

Les cas d'utilisation les plus simples pour les threadssurviennent lorsque nous avons plusieurs tâches complètement indépendantes que nous aimerions faire en même temps.

Par exemple, supposons que nous procédions au traitement du langage naturel sur un grand corpus de documents. On pourrait écrire une boucle :

```
fn process_files(filenames: Vec<String>) -> io::Result<()> {
    for document in filenames {
        let text = load(&document)?;  // read source file
        let results = process(text);  // compute statistics
        save(&document, results)?;    // write output file
    }
    Ok(())
}
```

Le programme s'exécute comme illustré à la Figure 19-1 .

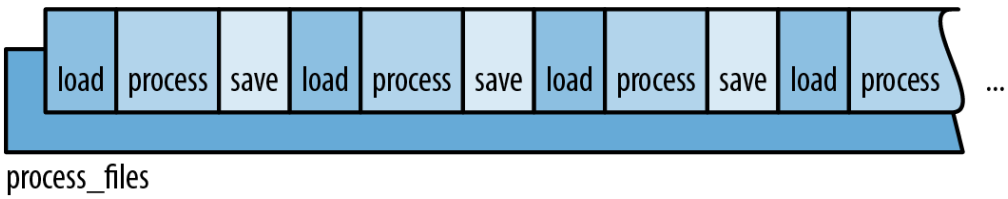

Image 19-1. Exécution monothread de `process_files()`

Étant donné que chaque document est traité séparément, il est relativement facile d'accélérer cette tâche en divisant le corpus en morceaux et en traitant chaque morceau sur un thread séparé, comme illustré à la Figure 19-2 .

Ce modèle est appelé *parallélisme fork-join.* Bifurquer , c'est *démarrer* un nouveau fil, et *rejoindre* un fil, c'est attendre qu'il se termine. Nous avons déjà vu cette technique : nous l'avons utilisée pour accélérer le programme Mandelbrot au chapitre 2 .

Le parallélisme fork-join est attrayant pour plusieurs raisons :

- C'est très simple. Fork-join est facile à mettre en œuvre et Rust facilite la mise en place.
- Cela évite les goulots d'étranglement. Il n'y a pas de verrouillage des ressources partagées dans fork-join. Le seul moment où un thread doit attendre un autre est à la fin. En attendant, chaque thread peut s'exécuter librement. Cela permet de réduire les frais généraux de commutation de tâches.
- Le calcul des performances est simple. Dans le meilleur des cas, en démarrant quatre threads, nous pouvons terminer notre travail en un quart de temps. La figure 19-2 montre une raison pour laquelle nous ne devrions pas nous attendre à cette accélération idéale : nous pourrions ne pas être en mesure de répartir le travail uniformément sur tous les threads. Une autre raison de prudence est que parfois les programmes de fork-join doivent passer un certain temps après la jointure des threads à *combiner* les résultats calculés par les threads. Autrement dit, isoler complètement les tâches peut entraîner un travail supplémentaire. Pourtant, en dehors de ces deux choses, tout programme lié au processeur avec des unités de travail isolées peut s'attendre à un coup de pouce significatif.
- Il est facile de raisonner sur l'exactitude du programme. Un programme fork-join est *déterministe* tant que les threads sont réellement isolés, comme les threads de calcul du programme Mandelbrot. Le programme produit toujours le même résultat, quelles que soient les variations de vitesse du fil. C'est un modèle de concurrence sans conditions de concurrence.

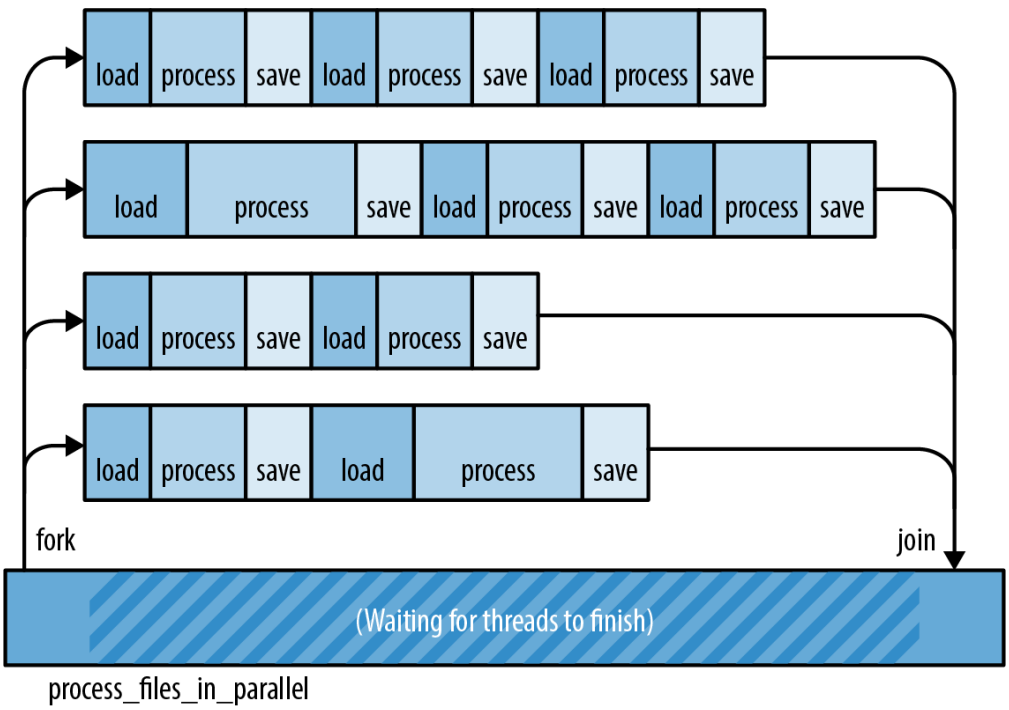

Image 19-2. Traitement de fichiers multithread à l'aide d'une approche fork-join

Le principal inconvénient du fork-join est qu'il nécessite des unités de travail isolées. Plus loin dans ce chapitre, nous examinerons certains problèmes qui ne se divisent pas aussi clairement.

Pour l'instant, restons-en à l'exemple du traitement du langage naturel. Nous allons montrer quelques manières d'appliquer le modèle fork-join à la `process_files` fonction.

frayer et rejoindre

La fonction `std::thread::spawn` lance un nouveau fil :

```
use std::thread;

thread::spawn(|| {
    println!("hello from a child thread");
});
```

Il prend un argument, une `FnOnce` fermeture ou une fonction. Rust démarre un nouveau thread pour exécuter le code de cette fermeture ou fonction. Le nouveau thread est un véritable thread du système d'exploitation avec sa propre pile, tout comme les threads en C++, C# et Java.

Voici un exemple plus substantiel, utilisant `spawn` pour implémenter une version parallèle de la `process_files` fonction d'avant :

```
use std::{thread, io};

fn process_files_in_parallel(filenames: Vec<String>) -> io:: Result<()> {
    // Divide the work into several chunks.
    const NTHREADS:usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }

    // Join: Wait for all threads to finish.
    for handle in thread_handles {
        handle.join().unwrap()?;
    }

    Ok(())
}
```

Prenons cette fonction ligne par ligne.

```
fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
```

Notre nouvelle fonction a la même signature de type que l'original `process_files`, ce qui en fait un remplacement pratique.

```
    // Divide the work into several chunks.
    const NTHREADS:usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);
```

Nous utilisons une fonction d'utilité `split_vec_into_chunks`, non illustrée ici, pour répartir le travail. Le résultat, `worklists`, est un vecteur de vecteurs. Il contient huit portions de taille égale du vecteur d'origine `filenames`.

```
    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }
```

Nous générons un fil pour chacun `worklist`. `spawn()` renvoie une valeur appelée a `JoinHandle`, que nous utiliserons plus tard. Pour l'instant, nous mettons tous les `JoinHandle` s dans un vecteur.

Notez comment nous obtenons la liste des noms de fichiers dans le thread de travail :

- `worklist` est défini et rempli par la `for` boucle, dans le thread parent.
- Dès que la `move` fermeture est créée, `worklist` est déplacé dans la fermeture.
- `spawn` puis déplace la fermeture (y compris le `worklist` vecteur) vers le nouveau thread enfant.

Ces déménagements ne coûtent pas cher. Comme les `Vec<String>` mouvements dont nous avons parlé au chapitre 4, les `String` s ne sont pas clonés. En fait, rien n'est alloué ou libéré. Les seules données déplacées sont elles- `Vec` mêmes : trois mots machine.

La plupart des threads que vous créez ont besoin à la fois de code et de données pour démarrer. Les fermetures de rouille, commodément, contiennent le code que vous voulez et les données que vous voulez.

Passons à autre chose :

```
        // Join: Wait for all threads to finish.
        for handle in thread_handles {
            handle.join().unwrap()?;
        }
```

Nous utilisons la `.join()` méthodedes `JoinHandle` s que nous avons collectés plus tôt pour attendre la fin des huit threads. Joindre des threads est souvent nécessaire pour l'exactitude, car un programme Rust se termine dès qu'il `main` revient, même si d'autres threads sont toujours en cours d'exécution. Les destructeurs ne sont pas appelés ; les threads supplémentaires sont simplement tués. Si ce n'est pas ce que vous voulez, assurez-vous de rejoindre tous les fils de discussion qui vous intéressent avant de revenir de `main` .

Si nous parvenons à traverser cette boucle, cela signifie que les huit threads enfants se sont terminés avec succès. Notre fonction se termine donc en retournant `Ok(())` :

```
        Ok(())
    }
```

Gestion des erreurs dans les threads

Le code que nous avons utilisé pour joindre les threads enfantsdans notre exemple est plus délicat qu'il n'y paraît, en raison de la gestion des erreurs. Reprenons cette ligne de code :

```
    handle.join().unwrap()?;
```

La `.join()` méthode fait deux choses intéressantes pour nous.

Tout d'abord, `handle.join()` les retoursc'est une `std::thread::Result` erreur *si le thread enfant a paniqué.* Cela rend le threading dans Rust considérablement plus robuste qu'en C++. En C++, un accès au tableau hors limites est un comportement indéfini, et il n'y a pas de protection du reste du système contre les conséquences. Dans Rust, la panique est sûre et par thread . Les frontières entre les threads servent de pare-feu pour la panique ; la panique ne se propage pas automatique-

ment d'un thread aux threads qui en dépendent. Au lieu de cela, une panique dans un thread est signalée comme une erreur `Result` dans d'autres threads. Le programme dans son ensemble peut facilement récupérer.

Dans notre programme, cependant, nous n'essayons pas de gérer la panique de façon fantaisiste. Au lieu de cela, nous avons immédiatementutiliser `.unwrap()` sur this `Result` , en affirmant qu'il s'agit d'un `Ok` résultat et non d'un `Err` résultat. Si un thread enfant panique , cette assertion échouera, de sorte que le thread parent paniquera également. Nous propageons explicitement la panique des threads enfants au thread parent.

Ensuite, `handle.join()` passe la valeur de retour du thread enfant au thread parent. La fermeture que nous avons passéeto `spawn` a un type de retour de `io::Result<()>` , car c'est ce qui `process_files` retourne. Cette valeur de retour n'est pas ignorée. Lorsque le thread enfant est terminé, sa valeur de retour est enregistrée et `JoinHandle::join()` transfère cette valeur au thread parent.

Le type complet renvoyé par `handle.join()` dans ce programme est `std::thread::Result<std::io::Result<()>>` . Le `thread::Result` fait partie de l' API `spawn` / ; `join` le `io::Result` fait partie de notre application.

Dans notre cas, après avoir déballé le `thread::Result` , nous utilisons l' `?` opérateur sur le `io::Result` , propageant explicitement les erreurs d'E/S des threads enfants au thread parent.

Tout cela peut sembler assez complexe. Mais considérez qu'il ne s'agit que d'une ligne de code, puis comparez cela avec d'autres langages. Le comportement par défaut en Java et C# consiste à envoyer les exceptions dans les threads enfants au terminal, puis à les oublier. En C++, la valeur par défaut consiste à abandonner le processus. Dans Rust, les erreurs sont des `Result` valeurs (données) au lieu d'exceptions (flux de contrôle). Ils sont livrés à travers les threads comme n'importe quelle autre valeur. Chaque fois que vous utilisez des API de threading de bas niveau, vous finissez par devoir écrire un code de gestion des erreurs minutieux, mais *étant donné que vous devez l'écrire,* `Result` c'est très agréable à avoir.

Partage de données immuables entre les threads

Supposons que l'analyseque nous faisons nécessite une grande base de données de mots et de phrases en anglais :

```
// before
fn process_files(filenames:Vec<String>)

// after
fn process_files(filenames: Vec<String>, glossary:&GigabyteMap)
```

Cela `glossary` va être important, nous le transmettons donc par référence. Comment pouvons-nous mettre à jour `process_files_in_parallel` pour transmettre le glossaire aux threads de travail ?

Le changement évident ne fonctionne pas :

```
fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: &GigabyteMap)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        thread_handles.push(
            spawn(move || process_files(worklist, glossary))  // error
        );
    }
    ...
}
```

Nous avons simplement ajouté un `glossary` argument à notre fonction et l'avons transmis à `process_files` . La rouille se plaint :

```
error: explicit lifetime required in the type of `glossary`
   |
38 |             spawn(move || process_files(worklist, glossary))  // error
   |             ^^^^^ lifetime `'static` required
```

Rust se plaint de la durée de vie de la fermeture que nous passons à `spawn` , et le message "utile" que le compilateur présente ici n'est en fait d'aucune aide.

`spawn` lance des threads indépendants. Rust n'a aucun moyen de savoir combien de temps le thread enfant s'exécutera, il suppose donc le pire : il suppose que le thread enfant peut continuer à fonctionner même après la fin du thread parent et que toutes les valeurs du thread parent ont disparu. De toute évidence, si le thread enfant doit durer aussi longtemps, la fermeture qu'il exécute doit également durer aussi longtemps. Mais cette fermeture a une durée de vie limitée : elle dépend de la référence `glossary` , et les références ne durent pas éternellement.

Notez que Rust a raison de rejeter ce code ! De la façon dont nous avons écrit cette fonction, il *est* possible qu'un thread rencontre une erreur d'E/S, ce qui provoque `process_files_in_parallel` un renflouement avant que les autres threads ne soient terminés. Les threads enfants pourraient finir par essayer d'utiliser le glossaire après que le thread principal l'ait libéré. Ce serait une course - avec un comportement indéfini comme prix, si le fil principal devait gagner. Rust ne peut pas permettre cela.

Il semble que ce `spawn` soit trop ouvert pour prendre en charge le partage de références entre les threads. En effet, on a déjà vu un cas comme celui-ci, dans « Closures That Steal » . Là, notre solution était de transférer la propriété des données au nouveau thread, en utilisant une `move` fermeture. Cela ne fonctionnera pas ici, car nous avons de nombreux threads qui doivent tous utiliser les mêmes données. Une alternative sûre est `clone` le glossaire complet pour chaque thread, mais comme il est volumineux, nous voulons éviter cela. Heureusement, la bibliothèque standard fournit un autre moyen : le comptage de références atomiques.

Nous avons décrit `Arc` dans « Rc et Arc : Propriété partagée » . Il est temps de l'utiliser :

```
use std:: sync::Arc;

fn process_files_in_parallel(filenames: Vec<String>,
                             glossary: Arc<GigabyteMap>)
    -> io::Result<()>
{
    ...
    for worklist in worklists {
        // This call to .clone() only clones the Arc and bumps the
        // reference count. It does not clone the GigabyteMap.
        let glossary_for_child = glossary.clone();
        thread_handles.push(
            spawn(move || process_files(worklist, &glossary_for_child))
        );
    }
    ...
}
```

Nous avons changé le type de `glossary` : pour exécuter l'analyse en parallèle, l'appelant doit passer dans an `Arc<GigabyteMap>` , un pointeur intelligent vers a `GigabyteMap` qui a été déplacé dans le tas, en utilisant `Arc::new(giga_map)` .

Lorsque nous appelons `glossary.clone()` , nous créons une copie du `Arc` pointeur intelligent, pas le tout `GigabyteMap` . Cela revient à incré-

menter un compteur de références.

Avec ce changement, le programme se compile et s'exécute, car il ne dépend plus des durées de vie des références. Tant qu'un *thread* possède un `Arc<GigabyteMap>`, il gardera la carte en vie, même si le thread parent se retire tôt. Il n'y aura pas de courses de données, car les données dans un `Arc` sont immuables.

Rayonne

La `spawn` fonction de la bibliothèque standard est une primitive importante, mais elle n'est pas conçue spécifiquement pour le parallélisme fork-join. De meilleures API de fork-join ont été construites dessus. Par exemple, au chapitre 2, nous avons utilisé la bibliothèque Crossbeam pour répartir du travail sur huit threads. *Filetages à portée* de Crossbeamprend en charge le parallélisme fork-join assez naturellement.

La rayonnebibliothèque, par Niko Matsakiset Josh Stone, est un autre exemple. Il fournit deux manières d'exécuter des tâches simultanément :

```
use rayon:: prelude::*;

// "do 2 things in parallel"
let (v1, v2) = rayon::join(fn1, fn2);

// "do N things in parallel"
giant_vector.par_iter().for_each(|value| {
    do_thing_with_value(value);
});
```

`rayon::join(fn1, fn2)` appelle simplement les deux fonctions et renvoie les deux résultats. La `.par_iter()` méthodecréea `ParallelIterator`, une valeur avec `map`, `filter` et d'autres méthodes, un peu comme un Rust `Iterator`. Dans les deux cas, Rayon utilise son propre pool de threads de travail pour répartir le travail lorsque cela est possible. Vous indiquez simplement à Rayon quelles tâches *peuvent* être effectuées en parallèle ; Rayon gère les fils et distribue le travail au mieux.

Les diagrammes de la Figure 19-3 illustrent deux manières de concevoir l'appel `giant_vector.par_iter().for_each(...)`. (a) Rayon agit comme s'il engendrait un thread par élément dans le vecteur. (b) Dans les coulisses, Rayon a un thread de travail par cœur de processeur, ce qui est plus efficace. Ce pool de threads de travail est partagé par tous les threads

de votre programme. Lorsque des milliers de tâches arrivent en même temps, Rayon divise le travail.

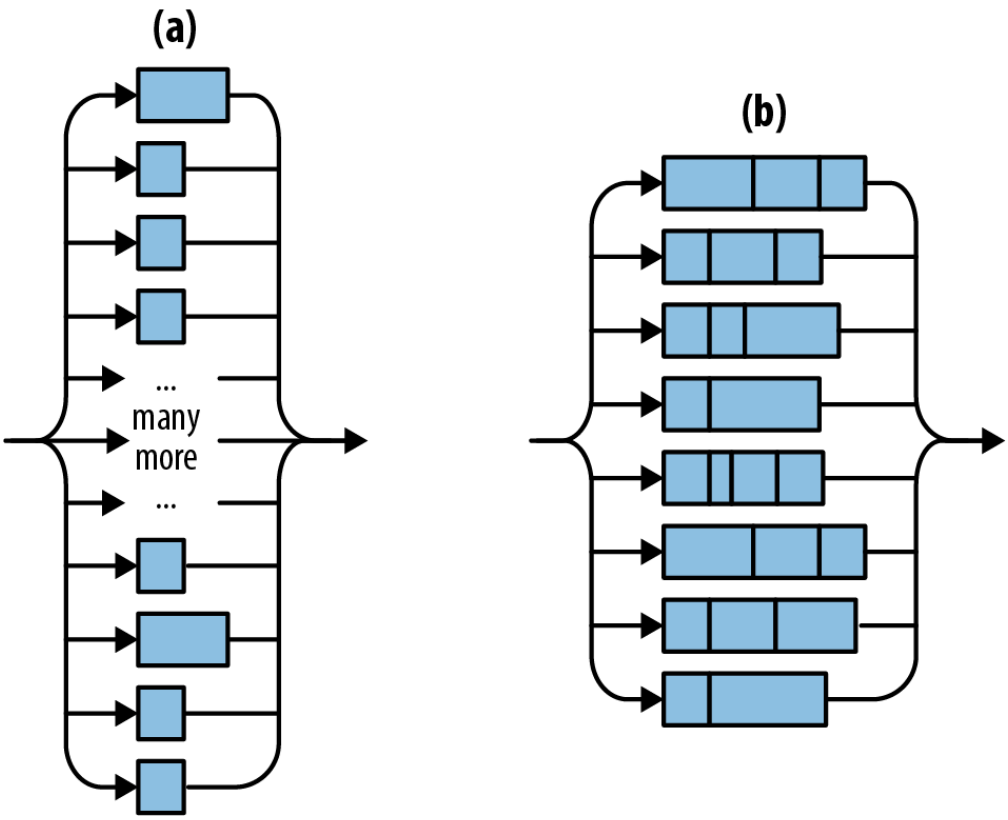

Image 19-3. Rayon en théorie et en pratique

Voici une version de `process_files_in_parallel` l'utilisation de Rayon et a `process_file` qui prend, plutôt que `Vec<String>` , juste un `&str` :

```
use rayon:: prelude::*;

fn process_files_in_parallel(filenames: Vec<String>, glossary: &GigabyteMap
    -> io::Result<()>
{
    filenames.par_iter()
        .map(|filename| process_file(filename, glossary))
        .reduce_with(|r1, r2| {
            if r1.is_err() { r1 } else { r2 }
        })
        .unwrap_or(Ok(()))
}
```

Ce code est plus court et moins délicat que la version utilisant `std::thread::spawn` . Regardons-le ligne par ligne :

- Tout d'abord, nous utilisons `filenames.par_iter()` pour créer un itérateur parallèle.
- Nous utilisons `.map()` pour appeler `process_file` chaque nom de fichier. Cela produit une `ParallelIterator` sur une séquence de

`io::Result<()>` valeurs.

- Nous utilisons `.reduce_with()` pour combiner les résultats. Ici, nous gardons la première erreur, le cas échéant, et supprimons le reste. Si nous voulions accumuler toutes les erreurs, ou les imprimer, nous pourrions le faire ici.
  La `.reduce_with()` méthode est également pratique lorsque vous passez une `.map()` fermeture qui renvoie une valeur utile en cas de succès. Ensuite, vous pouvez passer `.reduce_with()` une fermeture qui sait combiner deux résultats réussis.
- `reduce_with` renvoie un `Option` qui est `None` seulement si `filenames` était vide. Nous utilisons la `Option` méthode `.unwrap_or()` de pour faire le résultat `Ok(())` dans ce cas.

Dans les coulisses, Rayon équilibre dynamiquement les charges de travail entre les threads, en utilisant une technique appelée *vol de travail.* Il fera généralement un meilleur travail en gardant tous les processeurs occupés que nous ne pouvons le faire en divisant manuellement le travail à l'avance, comme dans "spawn and join" .

En prime, Rayon prend en charge le partage de références entre les threads. Tout traitement parallèle qui se produit dans les coulisses est assuré d'être terminé au `reduce_with` retour de l'heure. Cela explique pourquoi nous avons pu passer `glossary` à `process_file` même si cette fermeture sera appelée sur plusieurs threads.

(D'ailleurs, ce n'est pas un hasard si nous avons utilisé une `map` méthode et une `reduce` méthode. Le modèle de programmation MapReduce, popularisé par Google et Apache Hadoop, a beaucoup en commun avec le fork-join. Il peut être vu comme une approche fork-join pour interrogation de données distribuées.)

Revisiter l'ensemble de Mandelbrot

De retour au chapitre 2 , nous avons utilisé fork-joinconcurrence pour rendre l'ensemble de Mandelbrot. Cela a rendu le rendu quatre fois plus rapide - impressionnant, mais pas aussi impressionnant qu'il pourrait l'être, étant donné que le programme a généré huit threads de travail et l'a exécuté sur une machine à huit cœurs !

Le problème est que nous n'avons pas réparti la charge de travail équitablement. Calculer un pixel de l'image revient à exécuter une boucle (voir "Ce qu'est réellement l'ensemble de Mandelbrot"). Il s'avère que les parties gris pâle de l'image, où la boucle sort rapidement, sont beaucoup plus rapides à rendre que les parties noires, où la boucle exécute les 255 itérations complètes. Ainsi, bien que nous ayons divisé la zone en bandes hori-

zontales de taille égale, nous créions des charges de travail inégales, comme le montre la figure 19-4 .

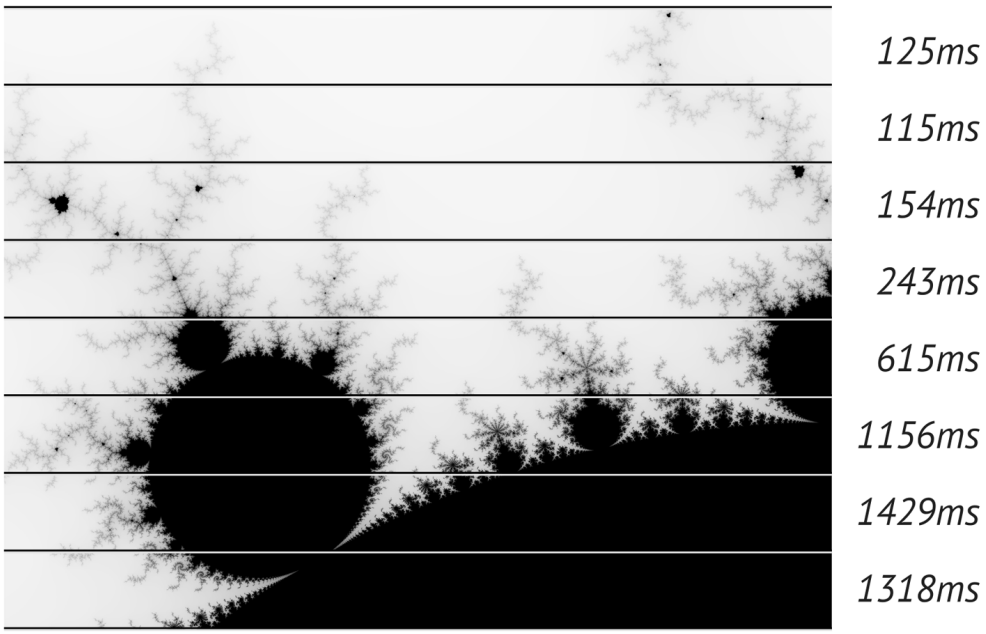

Image 19-4. Répartition inégale du travail dans le programme Mandelbrot

Ceci est facile à résoudre en utilisant Rayon. Nous pouvons simplement lancer une tâche parallèle pour chaque ligne de pixels dans la sortie. Cela crée plusieurs centaines de tâches que Rayon peut répartir sur ses threads. Grâce au vol de travail, peu importe que les tâches varient en taille. La rayonne équilibrera le travail au fur et à mesure.

Voici le code. La première ligne et la dernière ligne font partie de la `main` fonction que nous avons montrée dans "A Concurrent Mandelbrot Program" , mais nous avons changé le code de rendu, qui est tout entre les deux :

```
let mut pixels = vec![0; bounds.0 * bounds.1];

// Scope of slicing up `pixels` into horizontal bands.
{
    let bands:Vec<(usize, &mut [u8])> = pixels
        .chunks_mut(bounds.0)
        .enumerate()
        .collect();

    bands.into_par_iter()
        .for_each(|(i, band)| {
            let top = i;
            let band_bounds = (bounds.0, 1);
            let band_upper_left = pixel_to_point(bounds, (0, top),
                                                 upper_left, lower_right);
            let band_lower_right = pixel_to_point(bounds, (bounds.0, top +
                                                  upper_left, lower_right);
            render(band, band_bounds, band_upper_left, band_lower_right);
```

```
            });
    }

    write_image(&args[1], &pixels, bounds).expect("error writing PNG file");
```

Tout d'abord, nous créons `bands`, la collection de tâches que nous allons passer à Rayon. Chaque tâche est juste un tuple de type `(usize, &mut [u8])` : le numéro de ligne, puisque le calcul l'exige, et la tranche de `pixels` à remplir. Nous utilisons la `chunks_mut` méthode pour diviser le tampon d'image en lignes, `enumerate` pour attacher un numéro de ligne à chaque ligne, et `collect` pour avaler toutes les paires nombre-tranche dans un vecteur. (Nous avons besoin d'un vecteur car Rayon crée des itérateurs parallèles uniquement à partir de tableaux et de vecteurs.)

Ensuite, nous nous transformons `bands` en un itérateur parallèle et utilisons la `.for_each()` méthode pour dire à Rayon quel travail nous voulons faire.

Puisque nous utilisons Rayon, nous devons ajouter cette ligne à *main.rs* :

```
use rayon:: prelude::*;
```

et ceci à *Cargo.toml* :

```
[dépendances]
rayonne = "1"
```

Avec ces changements, le programme utilise désormais environ 7,75 cœurs sur une machine à 8 cœurs. C'est 75 % plus rapide qu'avant, lorsque nous divisons le travail manuellement. Et le code est un peu plus court, reflétant les avantages de laisser une caisse faire un travail (répartition du travail) plutôt que de le faire nous-mêmes.

Canaux

Un *canal*est un conduit unidirectionnel pour envoyer des valeurs d'un thread à un autre. En d'autres termes, il s'agit d'une file d'attente thread-safe.

La Figure 19-5 illustre l'utilisation des canaux. Ils sont quelque chose comme Unixpipes : une extrémité est destinée à l'envoi de données et l'autre à la réception. Les deux extrémités appartiennent généralement à deux threads différents. Mais alors que les canaux Unix servent à en-

voyer des octets, les canaux servent à envoyer des valeurs Rust. `sender.send(item)` place une seule valeur dans le canal ; `receiver.recv()` en supprime un. La propriété est transférée du thread d'envoi au thread de réception. Si le canal est vide, `receiver.recv()` bloque jusqu'à ce qu'une valeur soit envoyée.

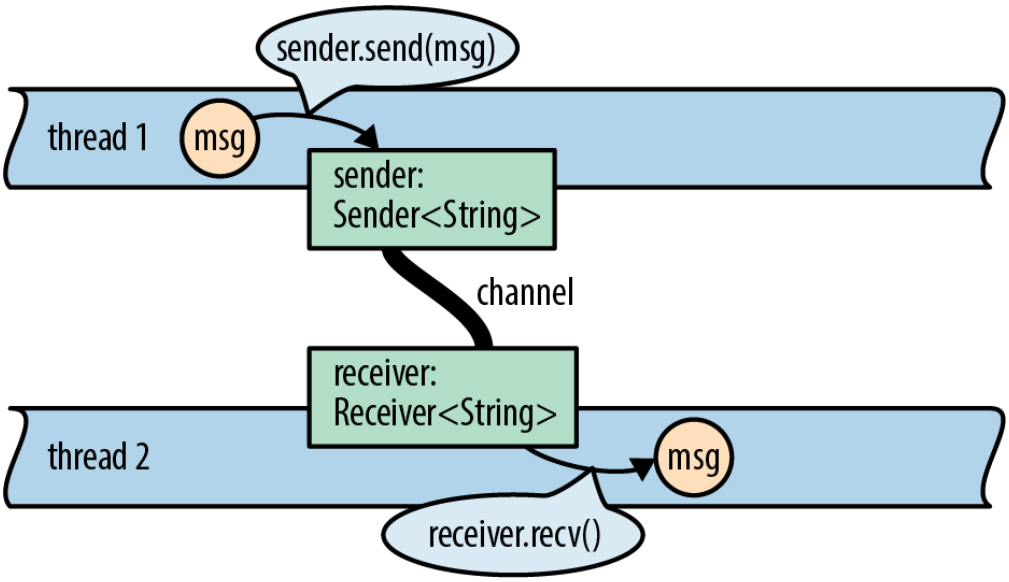

Image 19-5. Un canal pour `String` s : la propriété de la chaîne msg est transférée du thread 1 au thread 2.

Avec les canaux, les threads peuvent communiquer en se transmettant des valeurs. C'est un moyen très simple pour les threads de travailler ensemble sans utiliser de verrouillage ou de mémoire partagée.

Ce n'est pas une nouvelle technique. Erlang a des processus isolés et des messages transmis depuis 30 ans maintenant. Les tubes Unix existent depuis près de 50 ans. Nous avons tendance à penser que les canaux offrent de la flexibilité et de la composabilité, pas de la simultanéité, mais en fait, ils font tout ce qui précède. Un exemple de pipeline Unix est illustré à la Figure 19-6 . Il est certainement possible que les trois programmes fonctionnent en même temps.

Les canaux de rouille sont plus rapides que les tuyaux Unix. L'envoi d'une valeur la déplace plutôt que de la copier, et les déplacements sont rapides même lorsque vous déplacez des structures de données contenant de nombreux mégaoctets de données.

```
grep -h '^=' *.txt | sed 's/=//g' | sort
```

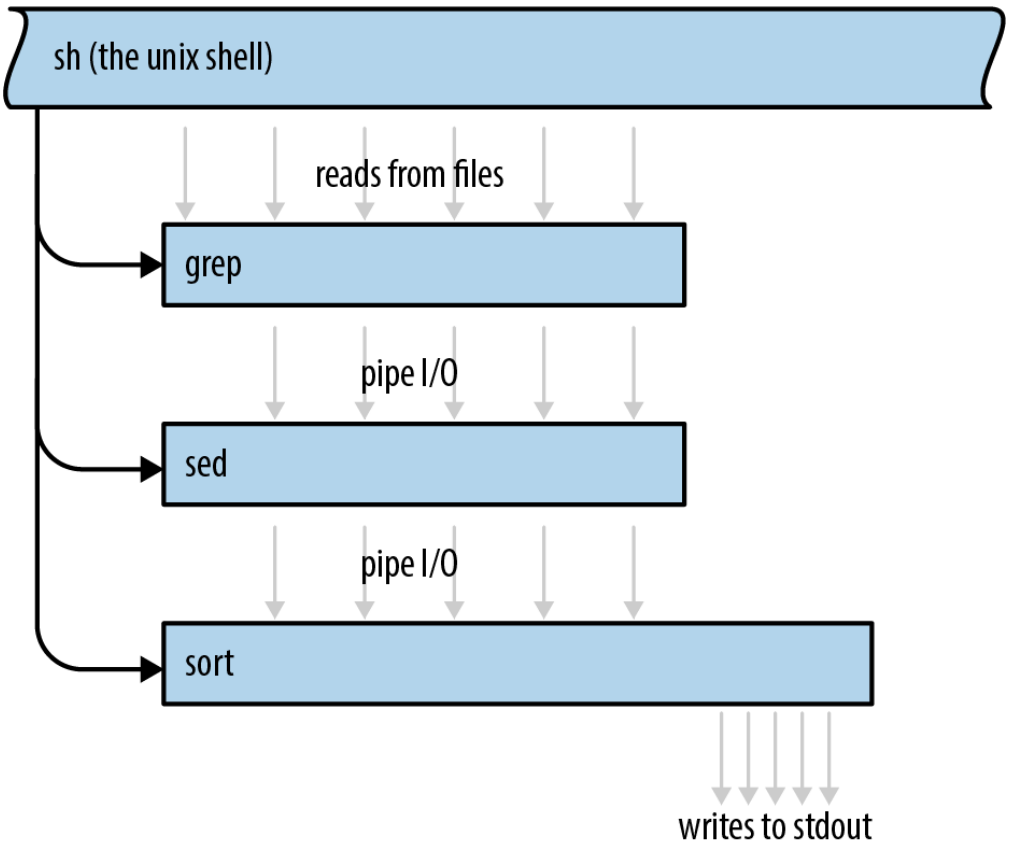

Illustration 19-6. Exécution d'un pipeline Unix

Envoi de valeurs

Plus dedans les prochaines sections, nous utiliserons des canaux pour construire un programme concurrent qui crée un *index inversé*, l'un des ingrédients clés d'un moteur de recherche. Chaque moteur de recherche travaille sur une collection particulière de documents. L'index inversé est la base de données qui indique quels mots apparaissent où.

Nous allons montrer les parties du code qui ont à voir avec les threads et les canaux. Le programme complet est court, environ un millier de lignes de code en tout.

Notre programme est structuré comme un pipeline, comme illustré à la Figure 19-7 . Les pipelines ne sont qu'une des nombreuses façons d'utiliser les canaux (nous aborderons quelques autres utilisations plus tard), mais ils constituent un moyen simple d'introduire la concurrence dans un programme monothread existant.

Nous utiliserons un total de cinq threads, chacun effectuant une tâche distincte. Chaque thread produit une sortie en continu pendant toute la durée de vie du programme. Le premier thread, par exemple, lit simplement les documents source du disque dans la mémoire, un par un. (Nous voulons qu'un thread le fasse car nous allons écrire ici le code le plus simple possible, en utilisant `fs::read_to_string` , qui est une API blo-

quante. Nous ne voulons pas que le processeur reste inactif lorsque le disque fonctionne.) La sortie de cette étape est long `String` par document, donc ce thread est connecté au thread suivant par un canal de `String` s.

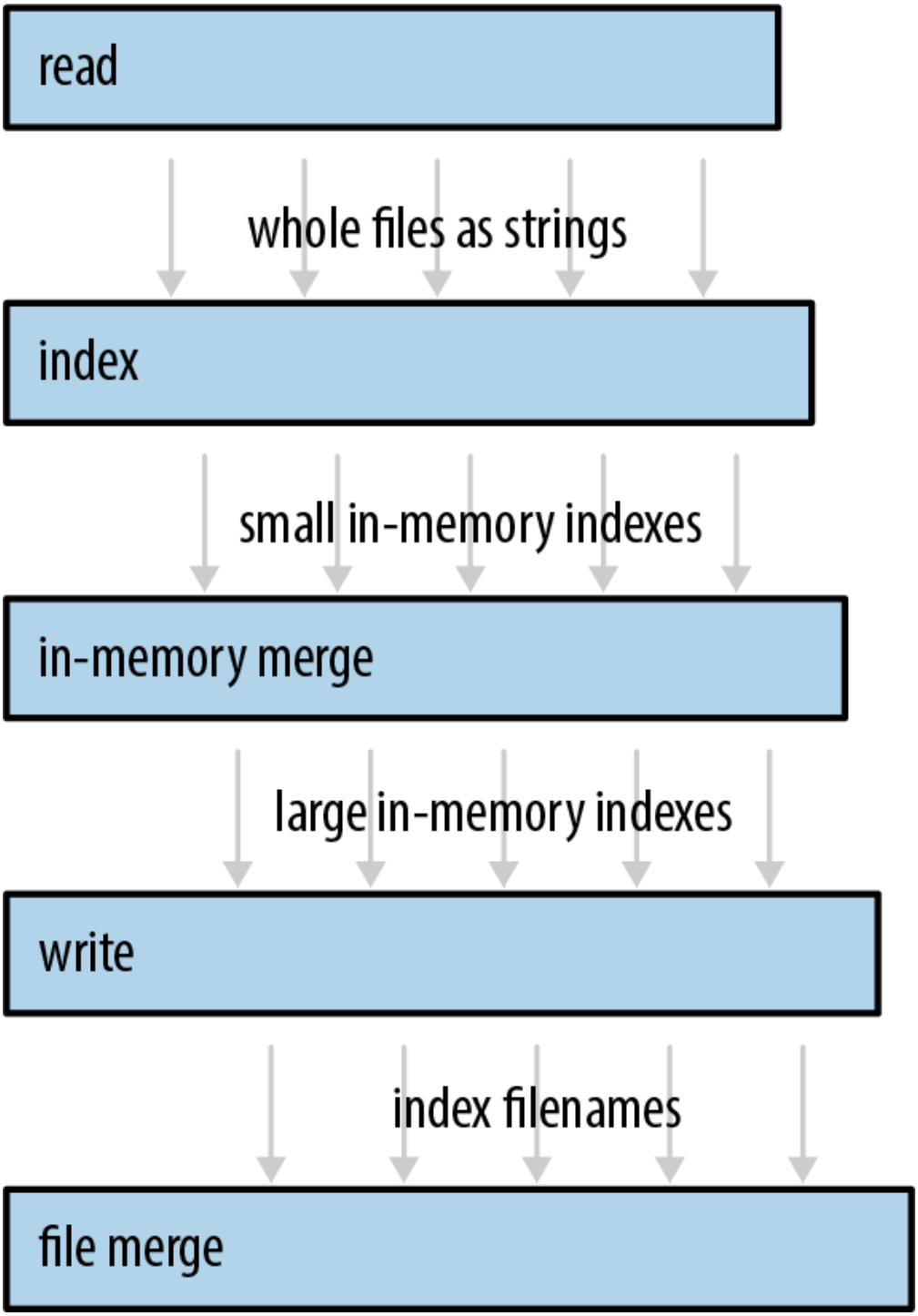

Image 19-7. Le pipeline de création d'index, où les flèches représentent les valeurs envoyées via un canal d'un thread à un autre (les E/S de disque ne sont pas affichées)

Notre programme commencera par générer le thread qui lit les fichiers. Supposons que `documents` est un `Vec<PathBuf>` , un vecteur de noms de fichiers. Le code pour démarrer notre fil de lecture de fichiers ressemble à ceci :

```
use std:: {fs, thread};
use std:: sync::mpsc;
```

```
    let (sender, receiver) = mpsc::channel();

    let handle = thread:: spawn(move || {
        for filename in documents {
            let text = fs::read_to_string(filename)?;

            if sender.send(text).is_err() {
                break;
            }
        }
        Ok(())
    });
```

Les chaînes font partie du `std::sync::mpsc` module. Nous expliquerons ce que signifie ce nom plus tard; Voyons d'abord comment fonctionne ce code. Nous commençons par créer un canal :

```
    let (sender, receiver) = mpsc::channel();
```

La `channel` fonction renvoie une paire de valeurs : un expéditeur et un destinataire. La structure de données de file d'attente sous-jacente est un détail d'implémentation que la bibliothèque standard n'expose pas.

Les canaux sont typés. Nous allons utiliser ce canal pour envoyer le texte de chaque fichier, nous avons donc un `sender` de type `Sender<String>` et un `receiver` de type `Receiver<String>` . Nous aurions pu demander explicitement un canal de chaînes, en écrivant `mpsc::channel::<String>()` . Au lieu de cela, nous laissons l'inférence de type de Rust le comprendre.

```
    let handle = thread::spawn(move || {
```

Comme précédemment, nous utilisons `std::thread::spawn` pour démarrer un fil. La propriété de `sender` (mais pas `receiver`) est transférée au nouveau fil via cette `move` fermeture.

Les quelques lignes de code suivantes lisent simplement les fichiers du disque :

```
        for filename in documents {
            let text = fs::read_to_string(filename)?;
```

Après avoir lu avec succès un fichier, nous envoyons son texte dans le canal :

```
            if sender.send(text).is_err() {
                break;
            }
        }
```

`sender.send(text)` déplace la valeur `text` dans le canal. En fin de compte, il sera à nouveau transféré à celui qui reçoit la valeur. Qu'elle `text` contienne 10 lignes de texte ou 10 mégaoctets, cette opération copie trois mots machine (la taille d'un `String` struct), et l' `receiver.recv()` appel correspondant copiera également trois mots machine.

Les méthodes `send` et `recv` renvoient toutes deux `Result` s, mais ces méthodes n'échouent que si l'autre extrémité du canal a été abandonnée. Un `send` appel échoue si le `Receiver` a été supprimé, car sinon la valeur resterait dans le canal pour toujours : sans un `Receiver` , aucun thread ne peut le recevoir. De même, un `recv` appel échoue s'il n'y a pas de valeurs en attente dans le canal et que le `Sender` a été supprimé, car sinon `recv` il attendrait indéfiniment : sans a `Sender` , il n'y a aucun moyen pour un thread d'envoyer la valeur suivante. Abandonner votre extrémité d'un canal est la façon normale de «raccrocher», de fermer la connexion lorsque vous en avez terminé.

Dans notre code, `sender.send(text)` n'échouera que si le thread du destinataire s'est terminé plus tôt. Ceci est typique pour le code qui utilise des canaux. Que cela se soit produit délibérément ou en raison d'une erreur, il est normal que notre fil de lecture se ferme tranquillement.

Lorsque cela se produit, ou que le thread finit de lire tous les documents, il renvoie `Ok(())` :

```
        Ok(())
    });
```

Notez que cette fermeture renvoie un `Result` . Si le thread rencontre une erreur d'E/S, il se ferme immédiatement et l'erreur est stockée dans le fichier `JoinHandle` .

Bien sûr, comme tout autre langage de programmation, Rust admet de nombreuses autres possibilités en matière de gestion des erreurs. Lorsqu'une erreur se produit, nous pouvons simplement l'imprimer à l'aide de `println!` et passer au fichier suivant. Nous pourrions transmettre les erreurs via le même canal que celui que nous utilisons pour les données, ce qui en ferait un canal de `Result` s ou créer un deuxième canal uni-

quement pour les erreurs. L'approche que nous avons choisie ici est à la fois légère et responsable : nous arrivons à utiliser l' `?` opérateur, donc il n'y a pas un tas de code passe-partout, ou même explicite `try/catch` comme vous pourriez le voir en Java, et pourtant les erreurs ne passeront pas silencieusement.

Pour plus de commodité, notre programme encapsule tout ce code dans une fonction qui renvoie à la fois le `receiver` (que nous n'avons pas encore utilisé) et le nouveau thread `JoinHandle` :

```
fn start_file_reader_thread(documents: Vec<PathBuf>)
    -> (mpsc:: Receiver<String>, thread:: JoinHandle<io:: Result<()>>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        ...
    });

    (receiver, handle)
}
```

Notez que cette fonction lance le nouveau thread et revient immédiatement. Nous allons écrire une fonction comme celle-ci pour chaque étape de notre pipeline.

Recevoir des valeurs

Maintenant nous avons un filexécuter une boucle qui envoie des valeurs. Nous pouvons générer un second thread exécutant une boucle qui appelle `receiver.recv()` :

```
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}
```

Mais `Receiver` les s sont itérables, il existe donc une meilleure façon d'écrire ceci :

```
for text in receiver {
    do_something_with(text);
}
```

Ces deux boucles sont équivalentes. Quelle que soit la façon dont nous l'écrivons, si le canal se trouve être vide lorsque le contrôle atteint le som-

met de la boucle, le thread récepteur se bloquera jusqu'à ce qu'un autre thread envoie une valeur. La boucle se terminera normalement lorsque le canal est vide et que le `Sender` a été supprimé. Dans notre programme, cela se produit naturellement lorsque le fil du lecteur se termine. Ce thread exécute une fermeture qui possède la variable `sender` ; lorsque la fermeture sort, `sender` est abandonné.

Nous pouvons maintenant écrire du code pour la deuxième étape du pipeline :

```
fn start_file_indexing_thread(texts: mpsc:: Receiver<String>)
    -> (mpsc:: Receiver<InMemoryIndex>, thread:: JoinHandle<()>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread:: spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() {
            let index = InMemoryIndex::from_single_document(doc_id, text);
            if sender.send(index).is_err() {
                break;
            }
        }
    });

    (receiver, handle)
}
```

Cette fonction génère un thread qui reçoit `String` des valeurs d'un canal (`texts`) et envoie des `InMemoryIndex` valeurs à un autre canal (`sender` / `receiver`). Le travail de ce thread consiste à prendre chacun des fichiers chargés lors de la première étape et à transformer chaque document en un petit index inversé en mémoire à un seul fichier.

La boucle principale de ce fil est simple. Tout le travail d'indexation d'un document est effectué par la fonction `InMemoryIndex::from_single_document` . Nous ne montrerons pas son code source ici, mais il divise la chaîne d'entrée aux limites des mots, puis produit une carte des mots aux listes de positions.

Cette étape n'effectue pas d'E/S, elle n'a donc pas à traiter avec `io::Error` s. Au lieu d'un `io::Result<()>` , il renvoie `()` .

Exécution du pipeline

Les trois étapes restantes sont de conception similaire. Chacun consomme un `Receiver` créé par l'étape précédente. Notre objectif pour le reste du

pipeline est de fusionner tous les petits index en un seul grand fichier d'index sur disque. Le moyen le plus rapide que nous ayons trouvé pour le faire est en trois étapes. Nous ne montrerons pas le code ici, juste les signatures de type de ces trois fonctions. La source complète est en ligne.

Tout d'abord, nous fusionnons les index en mémoire jusqu'à ce qu'ils deviennent peu maniables (étape 3) :

```
fn start_in_memory_merge_thread(file_indexes: mpsc:: Receiver<InMemoryIndex
    -> (mpsc:: Receiver<InMemoryIndex>, thread::JoinHandle<()>)
```

Nous écrivons ces grands index sur le disque (étape 4) :

```
fn start_index_writer_thread(big_indexes: mpsc:: Receiver<InMemoryIndex>,
                             output_dir: &Path)
    -> (mpsc:: Receiver<PathBuf>, thread:: JoinHandle<io::Result<()>>)
```

Enfin, si nous avons plusieurs fichiers volumineux, nous les fusionnons à l'aide d'un algorithme de fusion basé sur les fichiers (étape 5) :

```
fn merge_index_files(files: mpsc:: Receiver<PathBuf>, output_dir: &Path)
    -> io::Result<()>
```

Cette dernière étape ne renvoie pas de `Receiver`, car c'est la fin de la ligne. Il produit un seul fichier de sortie sur disque. Il ne renvoie pas de `JoinHandle`, car nous ne prenons pas la peine de créer un thread pour cette étape. Le travail est effectué sur le fil de l'appelant.

Nous arrivons maintenant au code qui lance les threads et vérifie les erreurs :

```
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf)
    -> io::Result<()>
{
    // Launch all five stages of the pipeline.
    let (texts,   h1) = start_file_reader_thread(documents);
    let (pints,   h2) = start_file_indexing_thread(texts);
    let (gallons, h3) = start_in_memory_merge_thread(pints);
    let (files,   h4) = start_index_writer_thread(gallons, &output_dir);
    let result = merge_index_files(files, &output_dir);

    // Wait for threads to finish, holding on to any errors that they encou
    let r1 = h1.join().unwrap();
    h2.join().unwrap();
    h3.join().unwrap();
    let r4 = h4.join().unwrap();
```

```
        // Return the first error encountered, if any.
        // (As it happens, h2 and h3 can't fail: those threads
        // are pure in-memory data processing.)
        r1?;
        r4?;
        result
    }
```

Comme précédemment, nous utilisons `.join().unwrap()` pour propager explicitement les paniques des threads enfants au thread principal. La seule autre chose inhabituelle ici est qu'au lieu d'utiliser `?` tout de suite, nous mettons de côté les `io::Result` valeurs jusqu'à ce que nous ayons rejoint les quatre threads.

Ce pipeline est 40 % plus rapide que l'équivalent à un seul thread. Ce n'est pas mal pour le travail d'un après-midi, mais dérisoire à côté du coup de pouce de 675% que nous avons obtenu pour le programme Mandelbrot. Nous n'avons clairement saturé ni la capacité d'E/S du système ni tous les cœurs du processeur. Que se passe-t-il?

Les pipelines sont comme des chaînes de montage dans une usine de fabrication : les performances sont limitées par le débit de l'étape la plus lente. Une toute nouvelle chaîne de montage non réglée peut être aussi lente que la production unitaire, mais les chaînes de montage récompensent un réglage ciblé. Dans notre cas, la mesure montre que la deuxième étape est le goulot d'étranglement. Notre thread d'indexation utilise `.to_lowercase()` and `.is_alphanumeric()`, il passe donc beaucoup de temps à fouiner dans les tables Unicode. Les autres étapes en aval de l'indexation passent la plupart de leur temps à dormir dans `Receiver::recv`, en attendant une entrée.

Cela signifie que nous devrions pouvoir aller plus vite. Au fur et à mesure que nous nous attaquerons aux goulots d'étranglement, le degré de parallélisme augmentera. Maintenant que vous savez comment utiliser les canaux et que notre programme est composé de morceaux de code isolés, il est facile de voir comment résoudre ce premier goulot d'étranglement. Nous pourrions optimiser manuellement le code pour la deuxième étape, comme n'importe quel autre code ; diviser le travail en deux étapes ou plus; ou exécuter plusieurs threads d'indexation de fichiers à la fois.

Fonctionnalités et performances de la chaîne

La `mpsc` partiede `std::sync::mpsc` signifie *multiproducteur, monoconsommateur*, une description laconique du type de communicationLes ca-

naux de Rust fournissent.

Les canaux de notre exemple de programme transportent des valeurs d'un seul émetteur vers un seul récepteur. C'est un cas assez courant. Mais les canaux Rust prennent également en charge plusieurs expéditeurs, au cas où vous auriez besoin, par exemple, d'un seul thread qui gère les demandes de plusieurs threads clients, comme illustré à la figure 19-8 .

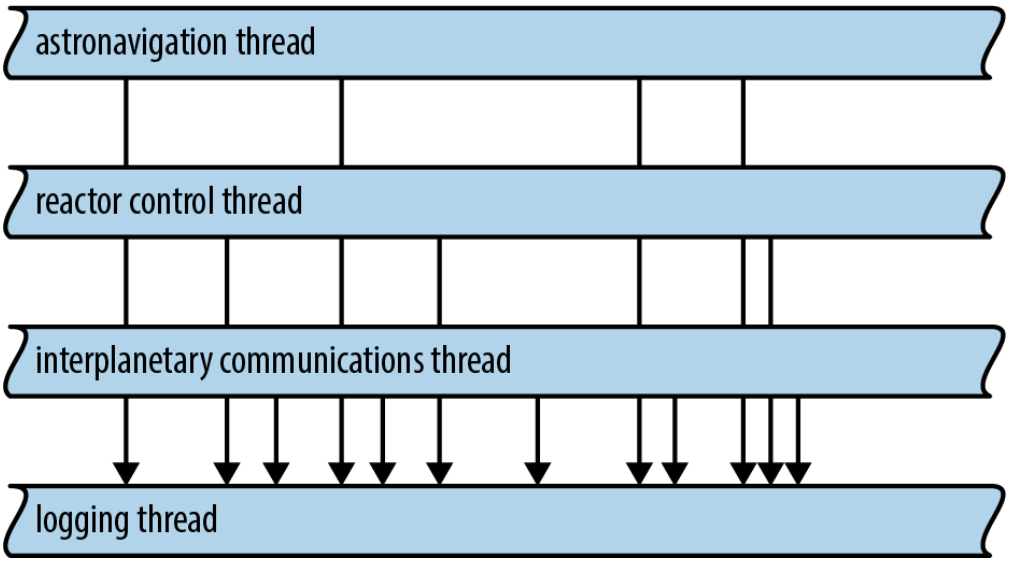

Image 19-8. Un canal unique recevant les requêtes de nombreux expéditeurs

`Sender<T>` implémente le `Clone` trait. Pour obtenir un canal avec plusieurs expéditeurs, créez simplement un canal normal et clonez l'expéditeur autant de fois que vous le souhaitez. Vous pouvez déplacer chaque `Sender` valeur vers un thread différent.

A `Receiver<T>` ne peut pas être cloné, donc si vous avez besoin de plusieurs threads recevant des valeurs du même canal, vous avez besoin d'un fichier `Mutex` . Nous montrerons comment le faire plus loin dans ce chapitre.

Les canaux de rouille sont soigneusement optimisés. Lorsqu'un canal est créé pour la première fois, Rust utilise une implémentation spéciale de file d'attente "one-shot". Si vous n'envoyez qu'un seul objet via le canal, la surcharge est minime. Si vous envoyez une deuxième valeur, Rust bascule vers une implémentation de file d'attente différente. Il s'installe sur le long terme, vraiment, préparant le canal à transférer de nombreuses valeurs tout en minimisant les frais généraux d'allocation. Et si vous clonez le `Sender` , Rust doit se rabattre sur une autre implémentation, qui est sûre lorsque plusieurs threads tentent d'envoyer des valeurs à la fois. Mais même la plus lente de ces trois implémentations est une file d'attente sans verrou, donc l'envoi ou la réception d'une valeur est au plus quelques opérations atomiques et une allocation de tas, plus le déplace-

ment lui-même. Les appels système ne sont nécessaires que lorsque la file d'attente est vide et que le thread récepteur doit donc se mettre en veille. Dans ce cas, bien sûr, le trafic via votre chaîne n'est pas maximisé de toute façon.

Malgré tout ce travail d'optimisation, il y a une erreur très facile à faire pour les applications concernant les performances du canal : envoyer des valeurs plus rapidement qu'elles ne peuvent être reçues et traitées. Cela provoque l'accumulation d'un arriéré de valeurs sans cesse croissant dans le canal. Par exemple, dans notre programme, nous avons constaté que le thread de lecture de fichiers (étape 1) pouvait charger des fichiers beaucoup plus rapidement que le thread d'indexation de fichiers (étape 2) ne pouvait les indexer. Le résultat est que des centaines de mégaoctets de données brutes seraient lues à partir du disque et placées dans la file d'attente en une seule fois.

Ce genre de mauvaise conduite coûte de la mémoire et blesse la localité. Pire encore, le thread d'envoi continue de fonctionner, utilisant le processeur et d'autres ressources système pour envoyer toujours plus de valeurs au moment où ces ressources sont le plus nécessaires du côté récepteur.

Ici, Rust reprend une page des pipes Unix. Unix utilise une astuce élégante pour fournir une *contre-pression* de sorte que les expéditeurs rapides sont obligés de ralentir : chaque canal sur un système Unix a une taille fixe, et si un processus essaie d'écrire dans un canal qui est momentanément plein, le système bloque simplement ce processus jusqu'à ce qu'il y ait de la place dans le canal. L'équivalent Rust est appelé *canal synchrone* :

```
use std:: sync::mpsc;

let (sender, receiver) = mpsc::sync_channel(1000);
```

Un canal synchrone est exactement comme un canal normal sauf que lorsque vous le créez, vous spécifiez le nombre de valeurs qu'il peut contenir. Pour un canal synchrone, `sender.send(value)` est potentiellement une opération bloquante. Après tout, l'idée est que le blocage n'est pas toujours mauvais. Dans notre exemple de programme, la modification de `channel` in `start_file_reader_thread` en a `sync_channel` with room for 32 values réduit l'utilisation de la mémoire de deux tiers sur notre ensemble de données de référence, sans diminuer le débit.

Sécurité des threads : envoyer et synchroniser

Jusqu'à présent, nous avons agicomme si toutes les valeurs pouvaient être librement déplacées et partagées entre les threads. C'est généralement vrai, mais l'histoire complète de la sécurité des threads de Rust repose sur deux traits intégrés, `std::marker::Send` et `std::marker::Sync` .

- Les types qui implémentent `Send` peuvent être passés en toute sécurité par valeur à un autre thread. Ils peuvent être déplacés d'un fil à l'autre.
- Les types qui implémentent `Sync` peuvent passer en toute sécurité par non `mut` -référence à un autre thread. Ils peuvent être partagés entre les threads.

Par *sûr* ici, nous entendons la même chose que nous entendons toujours : exempt de courses de données et d'autres comportements indéfinis.

Par exemple, dans l' `process_files_in_parallel` exemple, nous avons utilisé une fermeture pour passer a `Vec<String>` du thread parent à chaque thread enfant. Nous ne l'avons pas signalé à l'époque, mais cela signifie que le vecteur et ses chaînes sont alloués dans le thread parent, mais libérés dans le thread enfant. Le fait que `Vec<String>` implements `Send` est une promesse d'API que tout va bien : l'allocateur utilisé en interne par `Vec` et `String` est thread-safe.

(Si vous deviez écrire vos propres types `Vec` et `String` avec des répartiteurs rapides mais non sécurisés pour les threads, vous devriez les implémenter en utilisant des types qui ne sont pas `Send` , tels que des pointeurs non sécurisés. Rust en déduirait alors que vos types `NonThreadSafeVec` et ne le sont pas et les restreindrait à une utilisation à un seul thread. Mais c'est un cas rare.) `NonThreadSafeString Send`

Comme l'illustre la figure 19-9 , la plupart des types sont à la fois `Send` et `Sync` . Vous n'avez même pas besoin d'utiliser `#[derive]` pour obtenir ces traits sur les structures et les énumérations de votre programme. Rust le fait pour vous. Une structure ou une énumération est `Send` si ses champs sont `Send` , et `Sync` si ses champs sont `Sync` .

Certains types sont `Send` , mais pas `Sync` . C'est généralement fait exprès, comme dans le cas de `mpsc::Receiver` , où cela garantit que l'extrémité réceptrice d'un `mpsc` canal est utilisée par un seul thread à la fois.

Les quelques types qui ne sont ni `Send` ni `Sync` sont pour la plupart ceux qui utilisent la mutabilité d'une manière qui n'est pas thread-safe. Par exemple, considérons `std::rc::Rc<T>` , le type de pointeurs intelligents de comptage de références.

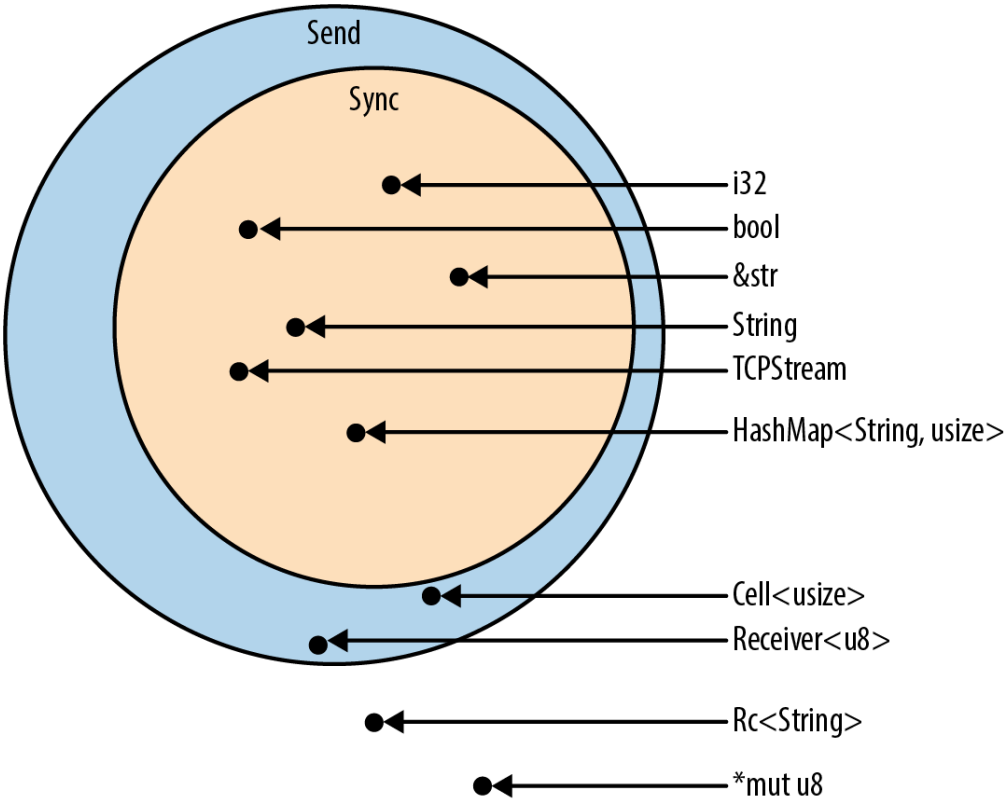

Illustration 19-9. `Send` et `Sync` types

Que se passerait-il si `Rc<String>` étaient `Sync` , permettant aux threads de partager un single `Rc` via des références partagées ? Si les deux threads essaient de cloner le `Rc` en même temps, comme illustré à la figure 19-10 , nous avons une course aux données car les deux threads incrémentent le nombre de références partagées. Le nombre de références peut devenir imprécis, entraînant un comportement d'utilisation après libération ou de double libération ultérieure, indéfini.

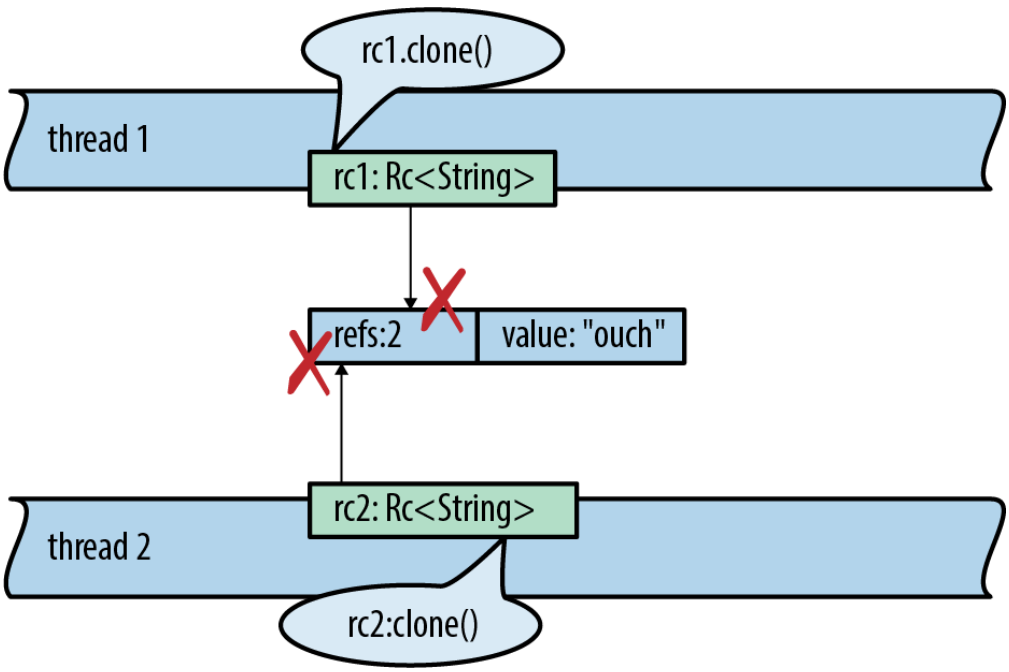

Figure 19-10. Pourquoi `Rc<String>` ni `Sync` ni `Send`

Bien sûr, Rust empêche cela. Voici le code pour configurer cette course aux données :

```
use std:: thread;
use std:: rc::Rc;

fn main() {
    let rc1 = Rc:: new("ouch".to_string());
    let rc2 = rc1.clone();
    thread::spawn(move || {  // error
        rc2.clone();
    });
    rc1.clone();
}
```

Rust refuse de le compiler, donnant un message d'erreur détaillé :

```
error: `Rc<String>` cannot be sent between threads safely
   |
10 |     thread::spawn(move || {  // error
   |             ^^^^^ `Rc<String>` cannot be sent between threads safely
   |
   = help: the trait `std::marker::Send` is not implemented for `Rc<String>
   = note: required because it appears within the type `[closure@...]`
   = note: required by `std::thread::spawn`
```

Vous pouvez maintenant voir comment `Send` et `Sync` aider Rust à appliquer la sécurité des threads. Ils apparaissent comme des limites dans la signature de type des fonctions qui transfèrent des données à travers les limites des threads. Lorsque vous `spawn` créez un thread, la fermeture que vous transmettez doit être `Send` , ce qui signifie que toutes les valeurs qu'il contient doivent être `Send` . De même, si vous souhaitez envoyer des valeursvia un canal vers un autre thread, les valeurs doivent être `Send` .

Canaliser presque n'importe quel itérateur vers un canal

Notre indice inverséle constructeur est construit comme un pipeline. Le code est assez clair, mais il nous permet de configurer manuellement des canaux et de lancer des threads. En revanche, les pipelines d'itérateurs que nous avons construits au chapitre 15 semblaient contenir beaucoup plus de travail dans quelques lignes de code seulement. Pouvons-nous construire quelque chose comme ça pour les pipelines de threads ?

En fait, ce serait bien si nous pouvions unifier les pipelines d'itérateurs et les pipelines de threads. Ensuite, notre constructeur d'index pourrait être écrit comme un pipeline d'itérateurs. Cela pourrait commencer ainsi :

```
documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender)    // filter out error results
    .off_thread()               // spawn a thread for the above work
    .map(make_single_file_index)
    .off_thread()               // spawn another thread for stage 2
    ...
```

Les traits nous permettent d'ajouter des méthodes aux types de bibliothèques standard, nous pouvons donc le faire. Nous commençons par écrire un trait qui déclare la méthode que nous voulons :

```
use std:: sync::mpsc;

pub trait OffThreadExt: Iterator {
    /// Transform this iterator into an off-thread iterator: the
    /// `next()` calls happen on a separate worker thread, so the
    /// iterator and the body of your loop run concurrently.
    fn off_thread(self) -> mpsc:: IntoIter<Self::Item>;
}
```

Ensuite, nous implémentons ce trait pour les types d'itérateurs. Cela aide qui `mpsc::Receiver` est déjà itérable:

```
use std::thread;

impl<T> OffThreadExt for T
    where T: Iterator + Send + 'static,
          T:: Item: Send + 'static
{
    fn off_thread(self) -> mpsc:: IntoIter<Self:: Item> {
        // Create a channel to transfer items from the worker thread.
        let (sender, receiver) = mpsc::sync_channel(1024);

        // Move this iterator to a new worker thread and run it there.
        thread::spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });

        // Return an iterator that pulls values from the channel.
        receiver.into_iter()
    }
}
```

La `where` clause de ce code a été déterminée via un processus similaire à celui décrit dans "Reverse-Engineering Bounds" . Au début, on avait juste ça :

```
impl<T> OffThreadExt for T
```

Autrement dit, nous voulions que l'implémentation fonctionne pour tous les itérateurs. Rust n'avait rien de tout cela. Parce que nous utilisons `spawn` pour déplacer un itérateur de type `T` vers un nouveau thread, nous devons spécifier `T: Iterator + Send + 'static` . Étant donné que nous renvoyons les éléments via un canal, nous devons spécifier `T::Item: Send + 'static` . Avec ces changements, Rust était satisfait.

Voici le caractère de Rust en quelques mots : nous sommes libres d'ajouter un outil puissant de concurrence à presque tous les itérateurs du langage, mais pas sans d'abord comprendre et documenter les restrictions qui le rendent sûr à utiliser..

Au-delà des pipelines

Dans cette section, nous avons utilisé des pipelines comme exemples, car les pipelines sont une manière agréable et évidente d'utiliser les canaux. Tout le monde les comprend. Ils sont concrets, pratiques et déterministes. Les canaux sont utiles pour plus que de simples pipelines, pourtant. Ils constituent également un moyen simple et rapide d'offrir n'importe quel service asynchrone à d'autres threads du même processus.

Par exemple, supposons que vous souhaitiez effectuer une journalisationsur son propre thread, comme dans la Figure 19-8 . D'autres threads pourraient envoyer des messages de journal au thread de journalisation sur un canal ; puisque vous pouvez cloner le canal `Sender` , de nombreux threads clients peuvent avoir des expéditeurs qui envoient des messages de journal au même thread de journalisation.

L'exécution d'un service comme la journalisation sur son propre thread présente des avantages. Le thread de journalisation peut faire pivoter les fichiers journaux chaque fois que nécessaire. Il n'a pas à faire de coordination sophistiquée avec les autres threads. Ces discussions ne seront pas bloquées. Les messages s'accumuleront sans danger dans le canal pendant un moment jusqu'à ce que le thread de journalisation se remette au travail.

Les canaux peuvent également être utilisés dans les cas où un thread envoie une requête à un autre thread et doit obtenir une sorte de réponse.

La requête du premier thread peut être une structure ou un tuple qui inclut un `Sender`, une sorte d'enveloppe auto-adressée que le deuxième thread utilise pour envoyer sa réponse. Cela ne signifie pas que l'interaction doit être synchrone. Le premier thread décide de bloquer et d'attendre la réponse ou d'utiliser la `.try_recv()` méthode pour l'interroger.

Les outils que nous avons présentés jusqu'à présent (fork-join pour un calcul hautement parallèle, canaux pour connecter de manière lâche des composants) sont suffisants pour un large éventail d'applications. Mais nous n'avons pas fini.

État mutable partagé

Dans les moisdepuis que vous avez publié la `fern_sim` caisse au chapitre 8, votre logiciel de simulation de fougère a vraiment décollé. Vous créez maintenant un jeu de stratégie multijoueur en temps réel dans lequel huit joueurs s'affrontent pour faire pousser des fougères d'époque pour la plupart authentiques dans un paysage jurassique simulé. Le serveur de ce jeu est une application massivement parallèle, avec des requêtes affluant sur de nombreux threads. Comment ces threads peuvent-ils se coordonner pour démarrer une partie dès que huit joueurs sont disponibles ?

Le problème à résoudre ici est que de nombreux threads ont besoin d'accéder à une liste partagée de joueurs qui attendent de rejoindre une partie. Ces données sont nécessairement modifiables et partagées entre tous les threads. Si Rust n'a pas d'état mutable partagé, où cela nous mène-t-il ?

Vous pouvez résoudre ce problème en créant un nouveau fil dont tout le travail consiste à gérer cette liste. D'autres threads communiqueraient avec lui via des canaux. Bien sûr, cela coûte un thread, ce qui entraîne une surcharge du système d'exploitation.

Une autre option consiste à utiliser les outils fournis par Rust pour partager en toute sécurité des données modifiables. De telles choses existent. Ce sont des primitives de bas niveau qui seront familières à tout programmeur système qui a travaillé avec des threads. Dans cette section, nous aborderons les mutex, les verrous en lecture/écriture, les variables de condition et les entiers atomiques. Enfin, nous montrerons comment implémenter des variables mutables globales dans Rust.

Qu'est-ce qu'un mutex ?

Un *mutex*(ou *lock*) est utilisé pour forcer plusieurs threads à se relayer lors de l'accès à certaines données. Nous présenterons les mutex de Rust dans la section suivante. Tout d'abord, il est logique de rappeler à quoi ressemblent les mutex dans d'autres langues. Une utilisation simple d'un mutexen C++ pourrait ressembler à ceci :

```
// C++ code, not Rust
void FernEmpireApp::JoinWaitingList(PlayerId player) {
    mutex.Acquire();

    waitingList.push_back(player);

    // Start a game if we have enough players waiting.
    if (waitingList.size() >= GAME_SIZE) {
        vector<PlayerId> players;
        waitingList.swap(players);
        StartGame(players);
    }

    mutex.Release();
}
```

Les appels `mutex.Acquire()` et `mutex.Release()` marquent le début et la fin d'une *section critique*dans ce code. Pour chacun `mutex` dans un programme, un seul thread peut être exécuté à la fois dans une section critique. Si un thread se trouve dans une section critique, tous les autres threads qui appellent `mutex.Acquire()` seront bloqués jusqu'à ce que le premier thread atteigne `mutex.Release()` .

On dit que le mutex *protège* les données : dans ce cas, `mutex` protects `waitingList` . Il est de la responsabilité du programmeur, cependant, de s'assurer que chaque thread acquiert toujours le mutex avant d'accéder aux données, et le libère ensuite.

Les mutex sont utiles pour plusieurs raisons :

- Ils empêchent *les courses de données,* situations où les threads de course lisent et écrivent simultanément la même mémoire. Les courses de données sont un comportement indéfini en C++ et Go. Les langages managés comme Java et C# promettent de ne pas planter, mais les résultats des courses aux données sont toujours (pour résumer) absurdes.
- Même si les courses de données n'existaient pas, même si toutes les lectures et écritures se produisaient une par une dans l'ordre du programme, sans mutex, les actions des différents threads pourraient s'entrelacer de manière arbitraire. Imaginez essayer d'écrire du code

qui fonctionne même si d'autres threads modifient ses données pendant son exécution. Imaginez essayer de le déboguer. Ce serait comme si votre programme était hanté.
- Les mutex prennent en charge la programmation avec *des invariants*, règles sur les données protégées qui sont vraies par construction lorsque vous les configurez et maintenues par chaque section critique.

Bien sûr, tout cela est vraiment la même raison : les conditions de course incontrôlées rendent la programmation insoluble. Les mutex apportent un peu d'ordre au chaos (mais pas autant d'ordre que les canaux ou les fork-join).

Cependant, dans la plupart des langues, les mutex sont très faciles à gâcher. En C++, comme dans la plupart des langages, les données et le verrou sont des objets distincts. Idéalement, les commentaires expliquent que chaque thread doit acquérir le mutex avant de toucher les données:

```
class FernEmpireApp {
    ...

private:
    // List of players waiting to join a game. Protected by `mutex`.
    vector<PlayerId> waitingList;

    // Lock to acquire before reading or writing `waitingList`.
    Mutex mutex;
    ...
};
```

Mais même avec des commentaires aussi gentils, le compilateur ne peut pas imposer un accès sécurisé ici. Lorsqu'un morceau de code néglige d'acquérir le mutex, nous obtenons un comportement indéfini. En pratique, cela signifie des bogues extrêmement difficiles à reproduire et à corriger.

Même en Java, où il existe une association théorique entre les objets et les mutex, la relation n'est pas très profonde. Le compilateur ne fait aucune tentative pour l'imposer, et en pratique, les données protégées par un verrou sont rarement exactement les champs de l'objet associé. Il inclut souvent des données dans plusieurs objets. Les schémas de verrouillage sont encore délicats. Les commentaires restent le principal outil pour les faire respecter.

Mutex<T>

Nous allons maintenant montrer une implémentationde la liste d'attente à Rust. Dans notre serveur de jeu Fern Empire, chaque joueur a un identifiant unique :

```
type PlayerId = u32;
```

La liste d'attente n'est qu'un ensemble de joueurs :

```
const GAME_SIZE:usize = 8;

/// A waiting list never grows to more than GAME_SIZE players.
type WaitingList = Vec<PlayerId>;
```

La liste d'attente est stockée sous la forme d'un champ de `FernEmpireApp`, un singleton configuré dans un `Arc` pendantdémarrage du serveur. Chaque thread a un `Arc` pointage vers lui. Il contient toute la configuration partagée et les autres flotsam dont notre programme a besoin. La plupart sont en lecture seule. La liste d'attente étant à la fois partagée et modifiable, elle doit être protégée par un `Mutex` :

```
use std:: sync::Mutex;

/// All threads have shared access to this big context struct.
struct FernEmpireApp {
    ...
    waiting_list:Mutex<WaitingList>,
    ...
}
```

Contrairement à C++, dans Rust, les données protégées sont stockées *dans* le fichier `Mutex`. Configurer les `Mutex` apparences comme ceci :

```
use std:: sync::Arc;

let app = Arc:: new(FernEmpireApp {
    ...
    waiting_list: Mutex::new(vec![]),
    ...
});
```

La création d'un nouveau `Mutex` ressemble à la création d'un nouveau `Box` ou `Arc`, mais tandis que `Box` et `Arc` signifient l'allocation de tas, `Mutex` il s'agit uniquement de verrouiller. Si vous voulez que votre `Mutex` être alloué dans le tas, vous devez le dire, comme nous l'avons fait ici en utilisant `Arc::new` pour l'ensemble de l'application et

`Mutex::new` justepour les données protégées. Ces types sont couramment utilisés ensemble : `Arc` est pratique pour partager des éléments entre les threads et `Mutex` est pratique pour les données mutables partagées entre les threads.

Nous pouvons maintenant implémenter la `join_waiting_list` méthode qui utilise le mutex :

```
impl FernEmpireApp {
    /// Add a player to the waiting list for the next game.
    /// Start a new game immediately if enough players are waiting.
    fn join_waiting_list(&self, player:PlayerId) {
        // Lock the mutex and gain access to the data inside.
        // The scope of `guard` is a critical section.
        let mut guard = self.waiting_list.lock().unwrap();

        // Now do the game logic.
        guard.push(player);
        if guard.len() == GAME_SIZE {
            let players = guard.split_off(0);
            self.start_game(players);
        }
    }
}
```

La seule façon d'accéder aux données est d'appeler la `.lock()` méthode:

```
let mut guard = self.waiting_list.lock().unwrap();
```

`self.waiting_list.lock()` blocs jusqu'à ce que le mutex puisse être obtenu. La `MutexGuard<WaitingList>` valeur renvoyée par cet appel de méthode est un wrapper mince autour d'un `&mut WaitingList`. Grâce aux coercitions deref, discutées, nous pouvons appeler des `WaitingList` méthodes directement sur la garde :

```
guard.push(player);
```

Le garde nous permet même d'emprunter des références directes aux données sous-jacentes. Le système de durée de vie de Rust garantit que ces références ne peuvent pas survivre à la garde elle-même. Il n'y a aucun moyen d'accéder aux données dans un `Mutex` sans détenir le verrou.

Lorsque `guard` est abandonné, le verrou est libéré. Normalement, cela se produit à la fin du bloc, mais vous pouvez également le déposer manuellement:

```
    if guard.len() == GAME_SIZE {
        let players = guard.split_off(0);
        drop(guard);  // don't keep the list locked while starting a game
        self.start_game(players);
    }
```

mut et mutex

Cela pourraitsembler étrange - certainement cela nous a semblé étrange au début - que notre `join_waiting_list` méthode ne prenne pas `self` par `mut` référence. Sa signature de type est :

```
fn join_waiting_list(&self, player:PlayerId)
```

La collection sous - jacente, `Vec<PlayerId>` , nécessite une référence `mut` lorsque vous appelez sa `push` méthode. Sa signature de type est :

```
pub fn push(&mut self, item:T)
```

Et pourtant, ce code compile et fonctionne correctement. Que se passe t-il ici?

En Rust, `&mut` signifie *accès exclusif* . Plain `&` signifie *un accès partagé* .

Nous sommes habitués à ce que les types transmettent `&mut` l'accès du parent à l'enfant, du conteneur au contenu. Vous vous attendez à ne pouvoir appeler des `&mut self` méthodes `starships[id].engine` que si vous avez une `&mut` référence à `starships` pour commencer (ou si vous possédez `starships` , auquel cas félicitations pour être Elon Musk). C'est la valeur par défaut, car si vous n'avez pas un accès exclusif au parent, Rust n'a généralement aucun moyen de s'assurer que vous avez un accès exclusif à l'enfant.

Mais `Mutex` a un moyen : la serrure. En fait, un mutex n'est guère plus qu'un moyen de faire exactement cela, de fournir *accès exclusif* (`mut`) aux données à l'intérieur, même si de nombreux threads peuvent avoir un accès *partagé* (non- `mut`) à lui- `Mutex` même.

Le système de type de Rust nous dit ce que `Mutex` fait. Il applique dynamiquement un accès exclusif, ce qui est généralement fait de manière statique, au moment de la compilation, par le compilateur Rust.

(Vous vous souviendrez peut-être que cela `std::cell::RefCell` fait la même chose, sauf sans essayer de prendre en charge plusieurs threads. `Mutex` et `RefCell` sont les deux saveurs de la mutabilité intérieure, que nous avons couvertes.)

Pourquoi les mutex ne sont pas toujours une bonne idée

Avant denous avons commencé par les mutex, nous avons présenté quelques approches de la concurrence qui auraient pu sembler étrangement faciles à utiliser correctement si vous venez de C++. Ce n'est pas une coïncidence : ces approches sont conçues pour fournir de solides garanties contre les aspects les plus déroutants de la programmation concurrente. Les programmes qui utilisent exclusivement le parallélisme fork-join sont déterministes et ne peuvent pas se bloquer. Les programmes qui utilisent des canaux se comportent presque aussi bien. Ceux qui utilisent des canaux exclusivement pour le pipelining, comme notre générateur d'index, sont déterministes : le moment de la livraison des messages peut varier, mais cela n'affectera pas la sortie. Etc. Les garanties sur les programmes multithreads sont sympas !

La conception de Rust `Mutex` vous fera presque certainement utiliser les mutex plus systématiquement et plus judicieusement que jamais auparavant. Mais cela vaut la peine de s'arrêter et de réfléchir à ce que les garanties de sécurité de Rust peuvent et ne peuvent pas aider.

Le code Safe Rust ne peut pas déclencher une *course aux données*, un type spécifique de bogue où plusieurs threads lisent et écrivent simultanément dans la même mémoire, produisant des résultats dénués de sens. C'est formidable : les courses de données sont toujours des bogues, et elles ne sont pas rares dans les vrais programmes multithreads.

Cependant, les threads qui utilisent des mutex sont sujets à d'autres problèmes que Rust ne résout pas pour vous :

- Les programmes Rust valides ne peuvent pas avoir de courses de données, mais ils peuvent toujours avoir d'autres *conditions de course* - des situations où le comportement d'un programme dépend de la synchronisation entre les threads et peut donc varier d'une exécution à l'autre. Certaines conditions de course sont bénignes. Certains se manifestent par des irrégularités générales et des bogues incroyablement difficiles à corriger. L'utilisation de mutex de manière non structurée invite à des conditions de concurrence. C'est à vous de vous assurer qu'ils sont bénins.

- L'état mutable partagé affecte également la conception du programme. Là où les canaux servent de limite d'abstraction dans votre code, ce qui facilite la séparation des composants isolés pour les tests, les mutex encouragent une méthode de travail "juste ajouter une méthode" qui peut conduire à une masse monolithique de code interdépendant.
- Enfin, les mutex ne sont tout simplement pas aussi simples qu'ils le paraissent au premier abord, comme le montreront les deux prochaines sections.

Tous ces problèmes sont inhérents aux outils. Utilisez une approche plus structurée lorsque vous le pouvez ; utilisez un `Mutex` quand vous le devez.

Impasse

Un thread peut se bloquerlui-même en essayant d'acquérir un verrou qu'il détient déjà :

```
let mut guard1 = self.waiting_list.lock().unwrap();
let mut guard2 = self.waiting_list.lock().unwrap();  // deadlock
```

Supposons que le premier appel à `self.waiting_list.lock()` réussisse, prenant le verrou. Le deuxième appel voit que le verrou est maintenu, il se bloque donc en attendant qu'il soit libéré. Il attendra pour toujours. Le thread en attente est celui qui détient le verrou.

Autrement dit, le verrou dans a `Mutex` n'est pas un verrou récursif.

Ici, le bug est évident. Dans un programme réel, les deux `lock()` appels peuvent être dans deux méthodes différentes, dont l'une appelle l'autre. Le code de chaque méthode, pris séparément, aurait l'air bien. Il existe également d'autres moyens d'obtenir un blocage, impliquant plusieurs threads qui acquièrent chacun plusieurs mutex à la fois. Le système d'emprunt de Rust ne peut pas vous protéger de l'impasse. La meilleure protection est de garder les sections critiques petites : entrez, faites votre travail et sortez.

Il est également possible d'obtenir un blocage avec les canaux. Par exemple, deux threads peuvent se bloquer, chacun attendant de recevoir un message de l'autre. Cependant, encore une fois, une bonne conception de programme peut vous donner une grande confiance que cela ne se produira pas dans la pratique. Dans un pipeline, comme notre générateur d'index inversé, le flux de données est acyclique. Un blocage est aussi improbable dans un tel programme que dans un pipeline shell Unix.

Mutex empoisonnés

`Mutex::lock()` renvoie un `Result` pour la même raisonqui `JoinHandle::join()` fait : échouer gracieusement si un autre thread a paniqué. Lorsque nous écrivons `handle.join().unwrap()` , nous disons à Rust de propager la panique d'un thread à l'autre. L'idiome `mutex.lock().unwrap()` est similaire.

Si un thread paniquetout en tenant un `Mutex` , Rust marque le `Mutex` comme *empoisonné.* Toute tentative ultérieure d' `lock` empoisonnement `Mutex` obtiendra un résultat d'erreur. Notre `.unwrap()` appel dit à Rust de paniquer si cela se produit, propageant la panique de l'autre thread à celui-ci.

À quel point est-ce mauvais d'avoir un mutex empoisonné ? Le poison semble mortel, mais ce scénario n'est pas nécessairement fatal. Comme nous l'avons dit au chapitre 7 , la panique est sans danger. Un thread paniqué laisse le reste du programme dans un état sûr.

La raison pour laquelle les mutex sont empoisonnés par la panique n'est donc pas la peur d'un comportement indéfini . Au contraire, le problème est que vous avez probablement programmé avec des invariants. Étant donné que votre programme a paniqué et est sorti d'une section critique sans terminer ce qu'il était en train de faire, peut-être après avoir mis à jour certains champs des données protégées mais pas d'autres, il est possible que les invariantssont maintenant cassés. La rouille empoisonne le mutex pour empêcher d'autres threads de tomber involontairement dans cette situation brisée et de l'aggraver. Vous *pouvez* toujours verrouiller un mutex empoisonné et accéder aux données qu'il contient, l'exclusion mutuelle étant pleinement appliquée ; voir la documentationpour `PoisonError::into_inner()` . Mais vous ne le ferez pas par accident.

Canaux multiconsommateurs utilisant des mutex

Nous avons mentionné plus tôt que les canaux de Rust sont multiplesproducteur, consommateur unique. Ou pour le dire plus concrètement, un canaln'en a qu'un `Receiver` . Nous ne pouvons pas avoir un pool de threads où de nombreux threads utilisent un seul `mpsc` canalsous forme de liste de travail partagée.

Cependant, il s'avère qu'il existe une solution de contournement très simple, en utilisant uniquement des éléments de bibliothèque standard. On peut ajouter un `Mutex` autour du `Receiver` et le partager quand même. Voici un module qui le fait :

```
pub mod shared_channel {
    use std:: sync:: {Arc, Mutex};
    use std:: sync:: mpsc::{channel, Sender, Receiver};

    /// A thread-safe wrapper around a `Receiver`.
    #[derive(Clone)]
    pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>>);

    impl<T> Iterator for SharedReceiver<T> {
        type Item = T;

        /// Get the next item from the wrapped receiver.
        fn next(&mut self) ->Option<T> {
            let guard = self.0.lock().unwrap();
            guard.recv().ok()
        }
    }

    /// Create a new channel whose receiver can be shared across threads.
    /// This returns a sender and a receiver, just like the stdlib's
    /// `channel()`, and sometimes works as a drop-in replacement.
    pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
        let (sender, receiver) = channel();
        (sender, SharedReceiver(Arc:: new(Mutex::new(receiver))))
    }
}
```

Nous utilisons un `Arc<Mutex<Receiver<T>>>` . Les génériques se sont vraiment accumulés. Cela se produit plus souvent en Rust qu'en C++. Cela peut sembler déroutant, mais souvent, comme dans ce cas, le simple fait de lire les noms peut aider à expliquer ce qui se passeactivé, comme illustré à la Figure 19-11 .

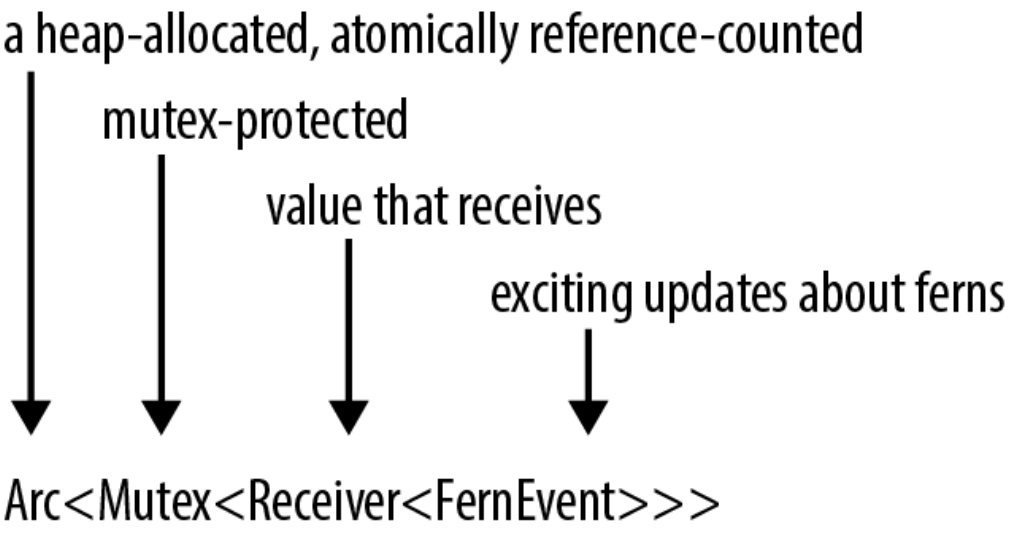

Image 19-11. Comment lire un type complexe

Verrouillages en lecture/écriture (RwLock<T>)

À présentpassons des mutex aux autres outils fournis dans `std::sync`, la boîte à outils de synchronisation des threads de la bibliothèque standard de Rust. Nous allons avancer rapidement, car une discussion complète de ces outils dépasse le cadre de ce livre.

Les programmes serveur ont souvent des informations de configuration qui sont chargées une seule fois et qui changent rarement. La plupart des threads interrogent uniquement la configuration, mais comme la configuration *peut* changer (il peut être possible de demander au serveur de recharger sa configuration à partir du disque, par exemple), elle doit de toute façon être protégée par un verrou. Dans de tels cas, un mutex peut fonctionner, mais c'est un goulot d'étranglement inutile. Les threads ne devraient pas avoir à interroger à tour de rôle la configuration si elle ne change pas. C'est un caspour un *verrou en lecture/écriture*, ou `RwLock`.

Alors queun mutex a une seule `lock` méthode, un verrou en lecture/écriture a deux méthodes de verrouillage, `read` et `write`. La `RwLock::write` méthode est comme `Mutex::lock`. Il attend un `mut` accès exclusif aux données protégées. La `RwLock::read` méthode fournit un non `mut`-accès, avec l'avantage qu'il est moins susceptible d'avoir à attendre, car de nombreux threads peuvent lire en toute sécurité à la fois. Avec un mutex, à un instant donné, les données protégées n'ont qu'un seul lecteur ou écrivain (ou aucun). Avec un verrou en lecture/écriture, il peut avoir un écrivain ou plusieurs lecteurs, un peu comme les références Rust en général.

`FernEmpireApp` peut avoir une structure pour la configuration, protégée par un `RwLock` :

```
use std:: sync::RwLock;

struct FernEmpireApp {
    ...
    config:RwLock<AppConfig>,
    ...
}
```

Les méthodes qui lisent la configuration utiliseraient `RwLock::read()` :

```
/// True if experimental fungus code should be used.
fn mushrooms_enabled(&self) ->bool {
    let config_guard = self.config.read().unwrap();
    config_guard.mushrooms_enabled
}
```

La méthode pour recharger la configuration utiliserait `RwLock::write()` :

```
fn reload_config(&self) -> io:: Result<()> {
    let new_config = AppConfig::load()?;
    let mut config_guard = self.config.write().unwrap();
    *config_guard = new_config;
    Ok(())
}
```

Rust, bien sûr, est particulièrement bien adapté pour faire respecter les règles de sécurité sur `RwLock` les données. Le concept d'écrivain unique ou de lecteur multiple est au cœur du système d'emprunt de Rust. `self.config.read()` renvoie une garde qui fournit `mut` un accès non (partagé) au `AppConfig` ; `self.config.write()` renvoie un autre type de garde qui fournit `mut` un accès (exclusif).

Variables de condition (Condvar)

Souvent, un thread doit attendre jusqu'à une certaine conditiondevient vrai :

- Lors de l'arrêt du serveur, le thread principal peut avoir besoin d'attendre que tous les autres threads aient fini de se fermer.
- Lorsqu'un thread de travail n'a rien à faire, il doit attendre qu'il y ait des données à traiter.
- Un thread implémentant un protocole de consensus distribué peut avoir besoin d'attendre qu'un quorum de pairs ait répondu.

Parfois, il existe une API de blocage pratique pour la condition exacte que nous voulons attendre, comme `JoinHandle::join` pour l'exemple d'arrêt du serveur. Dans d'autres cas, il n'y a pas d'API de blocage intégrée. Les programmes peuvent utiliser *des variables de condition* pour créer les leurs. Dans Rust, le `std::sync::Condvar` type implémente des variables de condition. A `Condvar` a des méthodes `.wait()` et `.notify_all()` ; `.wait()` bloque jusqu'à ce qu'un autre thread appelle `.notify_all()`.

Il y a un peu plus que cela, car une variable de condition concerne toujours une condition particulière vrai ou faux concernant certaines données protégées par un particulier `Mutex`. Ceci `Mutex` et le `Condvar` sont donc liés. Une explication complète est plus que ce que nous avons de place ici, mais pour le bénéfice des programmeurs qui ont déjà utilisé des variables de condition, nous montrerons les deux bits de code clés.

Lorsque la condition souhaitée devient vraie, nous appelons `Condvar::notify_all` (ou `notify_one`) pour réveiller tous les threads en attente :

```
self.has_data_condvar.notify_all();
```

Pour s'endormir et attendre qu'une condition devienne vraie, on utilise `Condvar::wait()` :

```
while !guard.has_data() {
    guard = self.has_data_condvar.wait(guard).unwrap();
}
```

Cette `while` boucle est un idiome standard pour les variables de condition. Cependant, la signature de `Condvar::wait` est inhabituelle. Il prend un `MutexGuard` objet par valeur, le consomme et renvoie un nouveau `MutexGuard` en cas de succès. Cela capture l'intuition que la `wait` méthode libère le mutex, puis le réacquiert avant de revenir. Passer la `MutexGuard` valeur par valeur est une façon de dire : « Je vous accorde, `.wait()` méthode, mon autorité exclusive pour libérer le mutex.

Atomique

Le `std::sync::atomic` modulecontient des types atomiques pour la programmation simultanée sans verrou. Ces types sont fondamentalement les mêmes que les atomiques C++ standard, avec quelques extras :

- `AtomicIsize` et `AtomicUsize` sont des types entiers partagés correspondant aux types à thread unique `isize` et `usize`.
- `AtomicI8`, `AtomicI16`, `AtomicI32`, `AtomicI64`, et leurs variantes non signées comme `AtomicU8` sont des types entiers partagés qui correspondent aux types à thread unique `i8`, `i16`, etc.
- An `AtomicBool` est une valeur partagée `bool`.
- An `AtomicPtr<T>` est une valeur partagée du type pointeur non sécurisé `*mut T`.

L'utilisation correcte des données atomiques dépasse le cadre de ce livre. Qu'il suffise de dire que plusieurs threads peuvent lire et écrire une valeur atomique à la fois sans provoquer de courses de données.

Au lieu des opérateurs arithmétiques et logiques habituels, les types atomiques exposent des méthodes qui effectuent *des opérations atomiques,* des chargements individuels, des magasins, des échanges et des opérations arithmétiques qui se produisent en toute sécurité, en tant qu'unité,

même si d'autres threads effectuent également des opérations atomiques qui touchent la même mémoire. emplacement. L'incrémentation d'un `AtomicIsize` nom `atom` ressemble à ceci :

```
use std:: sync:: atomic::{AtomicIsize, Ordering};

let atom = AtomicIsize:: new(0);
atom.fetch_add(1, Ordering::SeqCst);
```

Ces méthodes peuvent être compilées en instructions spécialisées en langage machine. Sur l'architecture x86-64, cet `.fetch_add()` appel se compile en une `lock incq` instruction, où un ordinaire `n += 1` pourrait se compiler en une `incq` instruction simple ou n'importe quel nombre de variations sur ce thème. Le compilateur Rust doit également renoncer à certaines optimisations autour de l'opération atomique, car, contrairement à un chargement ou à un stockage normal, il peut légitimement affecter ou être immédiatement affecté par d'autres threads.

L'argument `Ordering::SeqCst` est un *ordre de mémoire*. Les commandes de mémoire sont quelque chose comme les niveaux d'isolation des transactions dans une base de données. Ils indiquent au système à quel point vous vous souciez de notions philosophiques telles que les causes qui précèdent les effets et le temps qui n'a pas de boucles, par opposition à la performance. Les ordres de mémoire sont cruciaux pour l'exactitude du programme, et ils sont difficiles à comprendre et à raisonner. Heureusement, la pénalité de performances pour le choix de la cohérence séquentielle, l'ordre de mémoire le plus strict, est souvent assez faible, contrairement à la pénalité de performances pour la mise en mode d'une base de données SQL `SERIALIZABLE` . Alors en cas de doute, utilisez `Ordering::SeqCst` . Rust hérite de plusieurs autres commandes de mémoire de l'atomics C++ standard, avec diverses garanties plus faibles sur la nature de l'existence et de la causalité. Nous n'en discuterons pas ici.

Une utilisation simple de l'atome est pour l'annulation. Supposons que nous ayons un thread qui effectue des calculs de longue durée, comme le rendu d'une vidéo, et que nous aimerions pouvoir l'annuler de manière asynchrone. Le problème est de communiquer au thread que nous voulons qu'il se ferme. Nous pouvons le faire via un partage `AtomicBool` :

```
use std:: sync:: Arc;
use std:: sync:: atomic::AtomicBool;

let cancel_flag = Arc:: new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

Ce code crée deux `Arc<AtomicBool>` pointeurs intelligents qui pointent vers le même heap-allocated `AtomicBool`, dont la valeur initiale est `false`. Le premier, nommé `cancel_flag`, restera dans le thread principal. Le second, `worker_cancel_flag`, sera déplacé vers le thread de travail.

Voici le code du travailleur :

```
use std:: thread;
use std:: sync:: atomic::Ordering;

let worker_handle = thread:: spawn(move || {
    for pixel in animation.pixels_mut() {
        render(pixel); // ray-tracing - this takes a few microseconds
        if worker_cancel_flag.load(Ordering::SeqCst) {
            return None;
        }
    }
    Some(animation)
});
```

Après avoir rendu chaque pixel, le thread vérifie la valeur du drapeau en appelant sa `.load()` méthode :

```
worker_cancel_flag.load(Ordering::SeqCst)
```

Si dans le thread principal nous décidons d'annuler le thread de travail, nous stockons `true` dans le `AtomicBool` puis attendons que le thread se termine :

```
// Cancel rendering.
cancel_flag.store(true, Ordering::SeqCst);

// Discard the result, which is probably `None`.
worker_handle.join().unwrap();
```

Bien sûr, il existe d'autres façons de mettre cela en œuvre. Le `AtomicBool` ici pourrait être remplacé par un `Mutex<bool>` ou un canal. La principale différence est que les atomes ont un surcoût minimal. Les opérations atomiques n'utilisent jamais d'appels système. Un chargement ou un stockage se compile souvent en une seule instruction CPU.

Les atomiques sont une forme de mutabilité intérieure, comme `Mutex` ou `RwLock`, de sorte que leurs méthodes prennent `self` par référence (non-`mut`) partagée. Cela les rend utiles en tant que simples variables globales.

Variables globales

Supposernous écrivons du code réseau. On aimerait avoir une variable globale, un compteur qu'on incrémente à chaque fois qu'on sert un paquet :

```
/// Number of packets the server has successfully handled.
static PACKETS_SERVED:usize = 0;
```

Cela compile bien. Il n'y a qu'un seul problème. `PACKETS_SERVED` n'est pas modifiable, nous ne pouvons donc jamais le changer.

Rust fait tout ce qu'il peut raisonnablement faire pour décourager l'état mutable global. Les constantes déclarées avec `const` sont, bien sûr, immuables. Statiqueles variables sont également immuables par défaut, il n'y a donc aucun moyen d'obtenir une `mut` référence à une. A `static` peut être déclaré `mut` , mais y accéder n'est pas sûr. L'insistance de Rust sur la sécurité des threads est une raison majeure de toutes ces règles.

L'état mutable global a également des conséquences malheureuses sur le génie logiciel : il a tendance à rendre les différentes parties d'un programme plus étroitement couplées, plus difficiles à tester et plus difficiles à modifier ultérieurement. Pourtant, dans certains cas, il n'y a tout simplement pas d'alternative raisonnable, nous ferions donc mieux de trouver un moyen sûr de déclarer des variables statiques mutables.

Le moyen le plus simple de prendre en charge l'incrémentation `PACKETS_SERVED` , tout en le gardant thread-safe, est d'en faire un entier atomique:

```
use std:: sync:: atomic::AtomicUsize;

static PACKETS_SERVED: AtomicUsize = AtomicUsize::new(0);
```

Une fois ce statique déclaré, l'incrémentation du nombre de paquets est simple :

```
use std:: sync:: atomic::Ordering;

PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

Les globales atomiques sont limitées aux entiers simples et aux booléens. Cependant, créer une variable globale de n'importe quel autre type re-

vient à résoudre deux problèmes.

Tout d'abord, la variable doit être rendue thread-safe d'une manière ou d'une autre, car sinon elle ne peut pas être globale : pour des raisons de sécurité, les variables statiques doivent être à la fois `Sync` et non- `mut`. Heureusement, nous avons déjà vu la solution à ce problème. Rust a des types pour partager en toute sécurité des valeurs qui changent : `Mutex`, `RwLock` et les types atomiques. Ces types peuvent être modifiés même lorsqu'ils sont déclarés comme non- `mut`. C'est ce qu'ils font. (Voir "mut et Mutex".)

Deuxièmement, les initialiseurs statiques ne peuvent appeler que des fonctions spécifiquement marquées comme `const`, que le compilateur peut évaluer au moment de la compilation. Autrement dit, leur sortie est déterministe ; cela ne dépend que de leurs arguments, pas de tout autre état ou E/S. De cette façon, le compilateur peut intégrer les résultats de ce calcul en tant que constante de compilation. Ceci est similaireà C++ `constexpr`.

Les constructeurspour les `Atomic` types (`AtomicUsize`, `AtomicBool`, etc.) sonttoutes les `const` fonctions, ce qui nous a permis de créer un `static AtomicUsize` précédent. Quelques autres types, comme `String`, `Ipv4Addr` et `Ipv6Addr`, ont des constructeurs simples qui le sont `const` également.

Vous pouvez également définir vos propres `const` fonctions en préfixant simplement la signature de la fonction avec `const`. Rust limite ce que les `const` fonctions peuvent faire à un petit ensemble d'opérations, qui sont suffisantes pour être utiles tout en n'autorisant aucun résultat non déterministe. `const` les fonctions ne peuvent pas prendre des types comme arguments génériques, uniquement des durées de vie, et il n'est pas possible d'allouer de la mémoire ou d'opérer sur des pointeurs bruts, même dans des `unsafe` blocs. Nous pouvons cependant utiliser des opérations arithmétiques (y compris l'arithmétique d'enveloppement et de saturation), des opérations logiques qui ne court-circuitent pas et d'autres `const` fonctions. Par exemple, nous pouvons créer des fonctions pratiques pour faciliter la définition de `static` s et `const` s et réduire la duplication de code :

```
const fn mono_to_rgba(level: u8) -> Color {
    Color {
        red: level,
        green: level,
        blue: level,
        alpha:0xFF
    }
```

```
}

const WHITE: Color = mono_to_rgba(255);
const BLACK:Color = mono_to_rgba(000);
```

En combinant ces techniques, on pourrait être tenté d'écrire :

```
static HOSTNAME: Mutex<String> =
    Mutex:: new(String::new());  // error: calls in statics are limited to
                                 // constant functions, tuple structs, and
                                 // tuple variants
```

Malheureusement, tandis que `AtomicUsize::new()` et `String::new()` sont `const fn`, `Mutex::new()` n'est pas. Afin de contourner ces limitations, nous devons utiliser la `lazy_static` caisse.

Nous avons introduit la `lazy_static` caisse dans "Building Regex Values Lazily". Définir une variable avec la `lazy_static!` macro vous permet d'utiliser n'importe quelle expression pour l'initialiser ; il s'exécute la première fois que la variable est déréférencée et la valeur est enregistrée pour toutes les utilisations ultérieures.

Nous pouvons déclarer un global `Mutex`-controllé `HashMap` avec `lazy_static` comme ceci :

```
use lazy_static::lazy_static;

use std:: sync::Mutex;

lazy_static! {
    static ref HOSTNAME: Mutex<String> = Mutex:: new(String::new());
}
```

La même technique fonctionne pour d'autres structures de données complexes comme `HashMap`s et `Deque`s. C'est également très pratique pour les statiques qui ne sont pas modifiables du tout, mais nécessitent simplement une initialisation non triviale.

L'utilisation `lazy_static!` impose un coût de performance infime sur chaque accès aux données statiques. L'implémentation utilise `std::sync::Once`, une primitive de synchronisation de bas niveau conçue pour une initialisation unique. Dans les coulisses, chaque fois qu'un statique paresseux est accédé, le programme exécute une instruction de chargement atomique pour vérifier que l'initialisation a déjà eu lieu. (`Once` est un usage plutôt spécial, nous ne le couvrirons donc pas en

détail ici. Il est généralement plus pratique d'utiliser à la `lazy_static!` place. Cependant, il est pratique pour initialiser des bibliothèques non-Rust ; pour un exemple, voir "Une interface sécurisée pour libgit2" .)

À quoi ressemble le piratage de code concurrent dans Rust

Nous avons montré trois techniques d'utilisation des threads dans Rust : le parallélisme fork-join, les canaux et l'état mutable partagé avec des verrous. Notre objectif a été de fournir une bonne introduction aux éléments fournis par Rust, en mettant l'accent sur la manière dont ils peuvent s'intégrer dans de vrais programmes.

Rust insiste sur la sécurité, donc à partir du moment où vous décidez d'écrire un programme multithread, l'accent est mis sur la construction d'une communication sécurisée et structurée. Garder les threads principalement isolés est un bon moyen de convaincre Rust que ce que vous faites est sûr. Il se trouve que l'isolement est également un bon moyen de s'assurer que ce que vous faites est correct et maintenable. Encore une fois, Rust vous guide vers de bons programmes.

Plus important encore, Rust vous permet de combiner techniques et expériences. Vous pouvez itérer rapidement : discuter avec le compilateur vous permet d'être opérationnel correctement beaucoup plus rapidementque le débogage des courses de données.