Chapitre 20. Programmation asynchrone

Supposervous écrivez un serveur de chat. Pour chaque connexion réseau, il y a des paquets entrants à analyser, des paquets sortants à assembler, des paramètres de sécurité à gérer, des abonnements à des groupes de discussion à suivre, etc. Gérer tout cela pour de nombreuses connexions simultanément va demander une certaine organisation.

Idéalement, vous pourriez simplement démarrer un thread séparé pour chaque connexion entrante :

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

let listener = net:: TcpListener::bind(address)?;

for socket_result in listener.incoming() {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    thread::spawn(|| {
        log_error(serve(socket, groups));
    });
}
```

Pour chaque nouvelle connexion, cela génère un nouveau thread exécutant la `serve` fonction, qui est capable de se concentrer sur la gestion des besoins d'une seule connexion.

Cela fonctionne bien, jusqu'à ce que tout se passe bien mieux que prévu et que vous ayez soudainement des dizaines de milliers d'utilisateurs. Il n'est pas inhabituel que la pile d'un thread atteigne 100 Ko ou plus, et ce n'est probablement pas ainsi que vous souhaitez dépenser des gigaoctets de mémoire serveur. Les threads sont bons et nécessaires pour répartir le travail sur plusieurs processeurs, mais leurs besoins en mémoire sont tels que nous avons souvent besoin de moyens complémentaires, utilisés avec les threads, pour décomposer le travail.

Vous pouvez utiliser les tâches asynchrones Rustpour entrelacer de nombreuses activités indépendantes sur un seul thread ou un pool de threads de travail. Les tâches asynchrones sont similaires aux threads, mais sont beaucoup plus rapides à créer, se transmettent le contrôle entre elles plus efficacement et ont une surcharge de mémoire d'un ordre de grandeur

inférieur à celle d'un thread. Il est parfaitement possible d'avoir des centaines de milliers de tâches asynchrones exécutées simultanément dans un seul programme. Bien sûr, votre application peut toujours être limitée par d'autres facteurs tels que la bande passante du réseau, la vitesse de la base de données, le calcul ou les besoins en mémoire inhérents au travail, mais la surcharge de mémoire inhérente à l'utilisation des tâches est bien moins importante que celle des threads.

Généralement, le code Rust asynchrone ressemble beaucoup au code multithread ordinaire, sauf que les opérations susceptibles de bloquer, comme les E/S ou l'acquisition de mutex, doivent être traitées un peu différemment. Les traiter spécialement donne à Rust plus d'informations sur le comportement de votre code, ce qui rend possible l'amélioration des performances. La version asynchrone du code précédent ressemble à ceci :

```
use async_std::{net, task};

let listener = net:: TcpListener::bind(address).await?;

let mut new_connections = listener.incoming();
while let Some(socket_result) = new_connections.next().await {
    let socket = socket_result?;
    let groups = chat_group_table.clone();
    task::spawn(async {
        log_error(serve(socket, groups).await);
    });
}
```

Cela utilise la `async_std` caissemodules de mise en réseau et de tâches et ajoute `.await` après les appels qui peuvent bloquer. Mais la structure globale est la même que la version basée sur les threads .

L'objectif de ce chapitre n'est pas seulement de vous aider à écrire du code asynchrone, mais aussi de montrer comment il fonctionne avec suffisamment de détails pour que vous puissiez anticiper ses performances dans vos applications et voir où il peut être le plus utile.

- Pour montrer les mécanismes de la programmation asynchrone, nous exposons un ensemble minimal de fonctionnalités de langage qui couvre tous les concepts de base : les futurs, les fonctions asynchrones, les `await` expressions, les tâches et les `block_on` exécuteurs `spawn_local` .
- Ensuite, nous présentons des blocs asynchroneset l' `spawn` exécuteur testamentaire. Celles-ci sont essentielles pour faire un vrai travail,

mais conceptuellement, ce ne sont que des variantes des fonctionnalités que nous venons de mentionner. Au cours du processus, nous soulignons quelques problèmes que vous êtes susceptible de rencontrer et qui sont propres à la programmation asynchrone et expliquons comment les gérer.

- Pour montrer que toutes ces pièces fonctionnent ensemble, nous parcourons le code complet d'un serveur et d'un client de chat, dont le fragment de code précédent fait partie.
- Pour illustrer le fonctionnement des futurs primitifs et des exécuteurs, nous présentons des implémentations simples mais fonctionnelles de `spawn_blocking` et `block_on`.
- Enfin, nous expliquons le `Pin` type, qui apparaît de temps en temps dans les interfaces asynchrones pour garantir que les fonctions asynchrones et les blocs futurs sont utilisés en toute sécurité.

De synchrone à asynchrone

Envisagerque se passe-t-il lorsque vous appelez la fonction suivante (non asynchrone, complètement traditionnelle) :

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

fn cheapo_request(host: &str, port: u16, path: &str)
                      -> std:: io:: Result<String>
{
    let mut socket = net:: TcpStream::connect((host, port))?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host
    socket.write_all(request.as_bytes())?;
    socket.shutdown(net:: Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response)?;

    Ok(response)
}
```

Cela ouvre une connexion TCP à un serveur Web, lui envoie une requête HTTP simple dans un protocole obsolète, [1] puis lit la réponse. La figure 20-1 montre l'exécution de cette fonction dans le temps.

Ce diagramme montre comment la pile d'appels de fonction se comporte lorsque le temps s'écoule de gauche à droite. Chaque appel de fonction est une boîte, placée au-dessus de son appelant. Évidemment, la

`cheapo_request` fonction s'exécute tout au long de l'exécution. Il appelle des fonctions de la bibliothèque standard Rust comme `TcpStream::connect` les `TcpStream` implémentations de et de `write_all` et `read_to_string`. Ceux-ci appellent d'autres fonctions à leur tour, mais finalement le programme fait *des appels système*, demande au système d'exploitation de faire quelque chose, comme d'ouvrir une connexion TCP, ou de lire ou d'écrire des données.

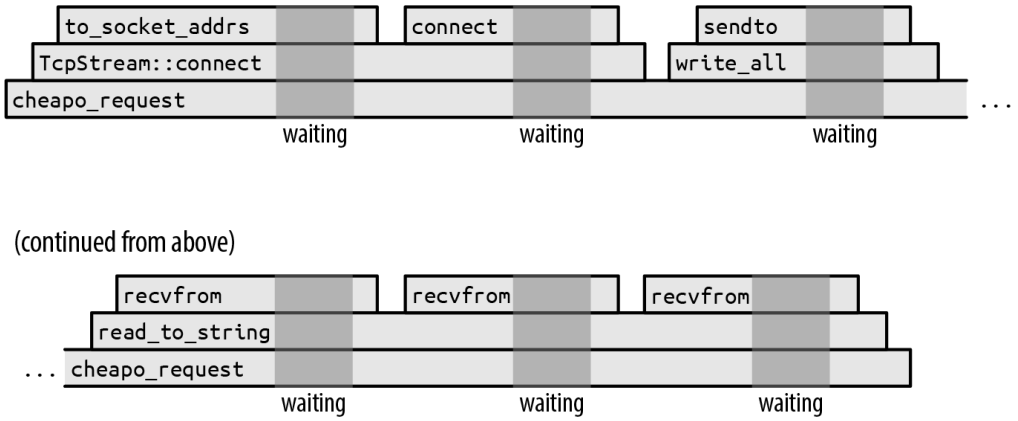

Illustration 20-1. Avancement d'une requête HTTP synchrone (des zones grises plus foncées attendent le système d'exploitation)

Les arrière-plans gris foncé marquent les moments où le programme attend que le système d'exploitation termine l'appel système. Nous n'avons pas dessiné ces temps à l'échelle. Si nous l'avions fait, tout le diagramme serait gris plus foncé : en pratique, cette fonction passe presque tout son temps à attendre le système d'exploitation. L'exécution du code précédent serait des fragments étroits entre les appels système.

Pendant que cette fonction attend le retour des appels système, son unique thread est bloqué : elle ne peut rien faire d'autre tant que l'appel système n'est pas terminé. Il n'est pas inhabituel que la pile d'un thread ait une taille de dizaines ou de centaines de kilo-octets, donc s'il s'agissait d'un fragment d'un système plus grand, avec de nombreux threads travaillant sur des tâches similaires, verrouiller les ressources de ces threads pour ne rien faire d'autre qu'attendre pourrait devenir assez cher.

Pour contourner ce problème, un thread doit pouvoir effectuer d'autres tâches en attendant que les appels système se terminent. Mais il n'est pas évident de savoir comment y parvenir. Par exemple, la signature de la fonction que nous utilisons pour lire la réponse du socket est :

```
fn read_to_string(&mut self, buf: &mut String) -> std:: io::Result<usize>;
```

C'est écrit directement dans le type : cette fonction ne revient pas tant que le travail n'est pas terminé ou que quelque chose ne va pas. Cette fonction est *synchrone* : l'appelant reprend lorsque l'opération est terminée. Si

nous voulons utiliser notre thread pour d'autres choses pendant que le système d'exploitation fait son travail, nous allons avoir besoin d'une nouvelle bibliothèque d'E/S qui fournit une version *asynchrone* de cette fonction.

Contrats à terme

L'approche de Rust pour prendre en charge les opérations asynchrones consiste à introduire un trait, `std::future::Future` :

```
trait Future {
    type Output;
    // For now, read `Pin<&mut Self>` as `&mut Self`.
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Outpu
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

A `Future` représente une opération dont vous pouvez tester l'achèvement. Une méthode future `poll` n'attend jamais la fin de l'opération : elle revient toujours immédiatement. Si l'opération est terminée, `poll` renvoie `Poll::Ready(output)`, où `output` est son résultat final. Sinon, ça revient `Pending`. Si et quand l'avenir vaut la peine d'être interrogé à nouveau, il promet de nous le faire savoir en invoquant un *réveil*, une fonction de rappel fournie dans le `Context`. Nous appelons cela le « modèle piñata » de la programmation asynchrone : la seule chose que vous pouvez faire avec un futur est de le frapper avec a `poll` jusqu'à ce qu'une valeur tombe.

Tous les systèmes d'exploitation modernes incluent des variantes de leurs appels système que nous pouvons utiliser pour implémenter ce type d'interface d'interrogation. Sous Unix et Windows, par exemple, si vous mettez un socket réseau en mode non bloquant, les lectures et les écritures renvoient une erreur si elles se bloquent ; vous devez réessayer plus tard.

Ainsi, une version asynchrone de `read_to_string` aurait une signature à peu près comme celle-ci :

```
fn read_to_string(&mut self, buf: &mut String)
    ->impl Future<Output = Result<usize>>;
```

C'est la même chose que la signature que nous avons montrée précédemment, à l'exception du type de retour : la version asynchrone renvoie *un futur de* a `Result<usize>` . Vous devrez interroger ce futur jusqu'à ce que vous en obteniez un `Ready(result)` . Chaque fois qu'il est interrogé, la lecture se poursuit aussi loin que possible. La finale `result` vous donne la valeur de succès ou une valeur d'erreur, tout comme une opération d'E/S ordinaire. C'est le modèle général : la version asynchrone de n'importe quelle fonction prend les mêmes arguments que la version synchrone, mais le type de retour est `Future` entouré d'un .

Appeler cette version de `read_to_string` ne lit en fait rien ; sa seule responsabilité est de construire et de rendre un avenir qui fera le vrai travail lorsqu'il sera interrogé. Ce futur doit contenir toutes les informations nécessaires pour mener à bien la demande formulée par l'appel. Par exemple, le futur renvoyé par this `read_to_string` doit mémoriser le flux d'entrée sur lequel il a été appelé et `String` auquel il doit ajouter les données entrantes. En fait, puisque le futur contient les références `self` et `buf` , la signature appropriée pour `read_to_string` doit être :

```
fn read_to_string<'a>(&'a mut self, buf: &'a mut String)
    ->impl Future<Output = Result<usize>> + 'a;
```

Cela ajoute des durées de vie pour indiquer que le futur renvoyé ne peut vivre que tant que les valeurs que `self` et `buf` empruntent.

La `async-std` caissefournit des versions asynchrones de toutes les fonctionnalités d' `std` E/S de , y compris un `Read` trait asynchrone avec une `read_to_string` méthode. `async-std` suit de près la conception de `std` , en réutilisant `std` les types de dans ses propres interfaces chaque fois que possible, de sorte que les erreurs, les résultats, les adresses réseau et la plupart des autres données associées sont compatibles entre les deux mondes. La connaissance de `std` vous aide à utiliser `async-std` , et vice versa.

L'une des règles du `Future` trait est qu'une fois qu'un futur est revenu `Poll::Ready` , il peut supposer qu'il ne sera plus jamais interrogé. Certains contrats à terme reviennent `Poll::Pending` pour toujours s'ils sont dépassés ; d'autres peuvent paniquer ou se bloquer. (Cependant, ils ne doivent pas violer la mémoire ou la sécurité des threads, ni provoquer un comportement indéfini.) L' `fuse` adaptateurLa méthode sur le `Future` trait transforme tout futur en un qui revient simplement `Poll::Pending` pour toujours. Mais toutes les façons habituelles de consommer des contrats à terme respectent cette règle, ce `fuse` n'est donc généralement pas nécessaire.

Si l'interrogation semble inefficace, ne vous inquiétez pas. L'architecture asynchrone de Rust est soigneusement conçue pour que, tant que vos fonctions d'E/S de base `read_to_string` sont correctement implémentées, vous n'interrogerez un avenir que lorsque cela en vaut la peine. Chaque fois `poll` qu'on l'appelle, quelque chose quelque part devrait revenir `Ready` , ou au moins faire des progrès vers cet objectif. Nous expliquerons comment cela fonctionne dans "Primitive Futures and Executors: When Is a Future Worth Polling Again? " .

Mais utiliser les contrats à terme semble être un défi : lorsque vous votez, que devez-vous faire lorsque vous obtenez `Poll::Pending` ? Vous devrez chercher d'autres travaux que ce fil peut faire pour le moment, sans oublier de revenir à ce futur plus tard et de le sonder à nouveau. L'ensemble de votre programme sera envahi par la plomberie qui gardera une trace de qui est en attente et de ce qui doit être fait une fois qu'ils seront prêts. La simplicité de notre `cheapo_request` fonction est ruinée.

Bonnes nouvelles! Ce n'est pas.

Fonctions asynchrones et expressions d'attente

Voici une version de `cheapo_request` écrite comme une *fonction asynchrone*:

```
use async_std:: io:: prelude:: *;
use async_std::net;

async fn cheapo_request(host: &str, port: u16, path: &str)
                                -> std:: io:: Result<String>
{
    let mut socket = net:: TcpStream::connect((host, port)).await?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host
    socket.write_all(request.as_bytes()).await?;
    socket.shutdown(net:: Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response).await?;

    Ok(response)
}
```

Ceci est jeton pour jeton identique à notre version originale, sauf:

- La fonction commence par `async fn` au lieu de `fn` .

- Il utilise les `async_std` versions asynchrones du crate de `TcpStream::connect`, `write_all` et `read_to_string`. Ceux-ci renvoient tous les futurs de leurs résultats. (Les exemples de cette section utilisent la version `1.7` de `async_std`.)
- Après chaque appel qui renvoie un futur, le code indique `.await`. Bien que cela ressemble à une référence à un champ struct nommé `await`, il s'agit en fait d'une syntaxe spéciale intégrée au langage pour attendre qu'un futur soit prêt. Une `await` expression est évaluée à la valeur finale du futur. C'est ainsi que la fonction obtient les résultats de `connect`, `write_all` et `read_to_string`.

Contrairement à une fonction ordinaire, lorsque vous appelez une fonction asynchrone, elle revient immédiatement, avant que le corps ne commence à s'exécuter. De toute évidence, la valeur de retour finale de l'appel n'a pas encore été calculée ; ce que vous obtenez est un *avenir de* sa valeur finale. Donc si vous exécutez ce code :

```
let response = cheapo_request(host, port, path);
```

alors `response` sera un futur de a `std::io::Result<String>`, et le corps de `cheapo_request` n'a pas encore commencé l'exécution. Vous n'avez pas besoin d'ajuster le type de retour d'une fonction asynchrone ; Rust traite automatiquement `async fn f(...) -> T` comme une fonction qui renvoie un futur de a `T`, pas a `T` directement.

Le futur renvoyé par une fonction asynchrone résume toutes les informations dont le corps de la fonction aura besoin pour s'exécuter : les arguments de la fonction, l'espace pour ses variables locales, etc. (C'est comme si vous aviez capturé le cadre de la pile de l'appel en tant que valeur Rust ordinaire.) So `response` doit contenir les valeurs transmises pour `host`, `port` et `path`, car `cheapo_request` le corps de va en avoir besoin pour s'exécuter.

Le type spécifique du futur est généré automatiquement par le compilateur, en fonction du corps et des arguments de la fonction. Ce type n'a pas de nom ; tout ce que vous en savez, c'est qu'il implémente `Future<Output=R>`, où `R` est le type de retour de la fonction asynchrone. En ce sens, les futurs des fonctions asynchrones sont comme des fermetures : les fermetures ont aussi des types anonymes, générés par le compilateur, qui implémentent les traits `FnOnce`, `Fn` et . `FnMut`

Lorsque vous interrogez pour la première fois le futur renvoyé par `cheapo_request`, l'exécution commence en haut du corps de la fonction et s'exécute jusqu'au premier `await` futur renvoyé par

`TcpStream::connect` . L' `await` expression interroge le `connect` futur, et s'il n'est pas prêt, alors il retourne `Poll::Pending` à son propre appelant : `cheapo_request` le futur de polling ne peut pas aller au-delà de ce premier `await` jusqu'à ce qu'un sondage du `TcpStream::connect` futur de retourne `Poll::Ready` . Ainsi, un équivalent approximatif de l'expression `TcpStream::connect(...).await` pourrait être :

```
{
    // Note: this is pseudocode, not valid Rust
    let connect_future = TcpStream:: connect(...);
    'retry_point:
     match connect_future.poll(cx) {
        Poll:: Ready(value) => value,
        Poll:: Pending => {
            // Arrange for the next `poll` of `cheapo_request`'s
            // future to resume execution at 'retry_point.
            ...
            return Poll::Pending;
        }
    }
}
```

Une `await` expressions'approprie l'avenir, puis l'interroge. S'il est prêt, la valeur finale du futur est la valeur de l' `await` expression et l'exécution continue. Sinon, il renvoie le `Poll::Pending` à son propre appelant.

Mais surtout, la prochaine interrogation du `cheapo_request` futur de ne recommence pas au sommet de la fonction : à la place, elle *reprend* l' exécution en cours de fonction au point où elle est sur le point d'interroger `connect_future` . Nous ne progressons pas vers le reste de la fonction asynchrone tant que ce futur n'est pas prêt.

Au fur et à mesure que `cheapo_request` le futur de continue d'être interrogé, il se fraye un chemin à travers le corps de la fonction de l'un `await` à l'autre, ne progressant que lorsque le sous-futur qu'il attend est prêt. Ainsi, le nombre de fois où `cheapo_request` le futur de doit être interrogé dépend à la fois du comportement des sous-futurs et du propre flux de contrôle de la fonction. `cheapo_request` Le futur de suit le point auquel le prochain `poll` devrait reprendre, et tout l'état local - variables, arguments, temporaires - dont la reprise aura besoin.

La possibilité de suspendre l'exécution en cours de fonctionnement, puis de reprendre plus tard est unique aux fonctions asynchrones. Lorsqu'une fonction ordinaire revient, son cadre de pile disparaît pour de bon. Étant donné que `await` les expressions dépendent de la possibilité de re-

prendre, vous ne pouvez les utiliser qu'à l'intérieur des fonctions asynchrones.

Au moment d'écrire ces lignes, Rust n'autorise pas encore les traits à avoir des méthodes asynchrones. Seules les fonctions libres et les fonctions inhérentes à un type spécifique peuvent être asynchrones. La levée de cette restriction nécessitera un certain nombre de modifications de la langue. En attendant, si vous avez besoin de définir des caractéristiques qui incluent des fonctions asynchrones, envisagez d'utiliser la `async-trait` caisse, qui fournit une solution de contournement basée sur des macros.

Appel de fonctions asynchrones à partir de code synchrone : block_on

Dans un sens, les fonctions asynchrones se renvoient la balle. Certes, il est facile d'obtenir la valeur d'un futur dans une fonction asynchrone : juste `await` ça. Mais la fonction async *elle-même* renvoie un futur, c'est donc maintenant à l'appelant de faire l'interrogation d'une manière ou d'une autre. En fin de compte, quelqu'un doit attendre une valeur.

Nous pouvons appeler `cheapo_request` à partir d'une fonction synchrone ordinaire (comme `main` , par exemple) en utilisant `async_std` la `task::block_on` fonction de , qui prend un futur et l'interroge jusqu'à ce qu'il produise une valeur :

```
fn main() -> std:: io:: Result<()> {
    use async_std::task;

    let response = task::block_on(cheapo_request("example.com", 80, "/"))?;
    println!("{}", response);
    Ok(())
}
```

Puisqu'il `block_on` s'agit d'une fonction synchrone qui produit la valeur finale d'une fonction asynchrone, vous pouvez la considérer comme un adaptateur du monde asynchrone au monde synchrone. Mais son caractère bloquant signifie également que vous ne devriez jamais l'utiliser `block_on` dans une fonction asynchrone : cela bloquerait tout le thread jusqu'à ce que la valeur soit prête. Utilisez à la `await` place.

La figure 20-2 montre une exécution possible de `main` .

La chronologie supérieure, "Vue simplifiée", montre une vue abstraite des appels asynchrones du programme : `cheapo_request` premiers appels `TcpStream::connect` pour obtenir un socket, puis appels `write_all` et `read_to_string` sur ce socket. Puis ça revient. Ceci est très similaire à la chronologie de la version synchrone du `cheapo_request` début de ce chapitre.

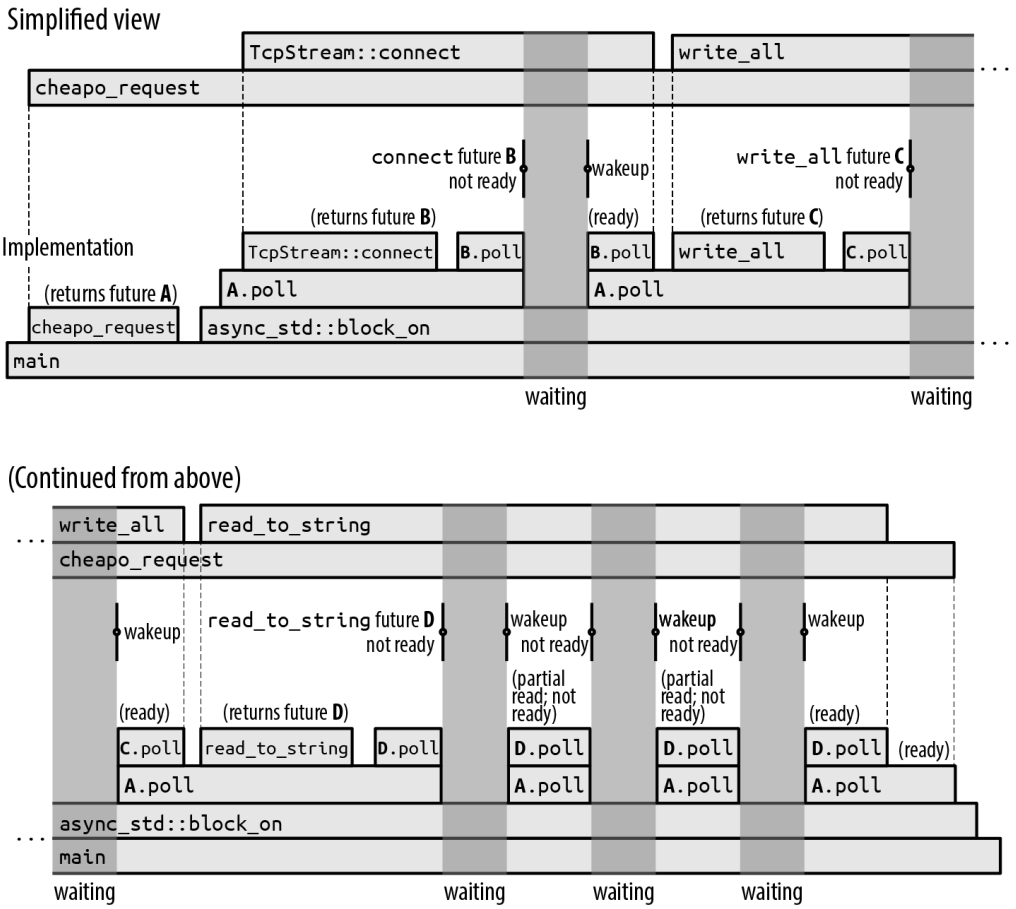

Illustration 20-2. Blocage sur une fonction asynchrone

Mais chacun de ces appels asynchrones est un processus en plusieurs étapes : un futur est créé puis interrogé jusqu'à ce qu'il soit prêt, créant et interrogeant peut-être d'autres sous-futurs au cours du processus. La chronologie inférieure, « Implémentation », affiche les appels synchrones réels qui implémentent ce comportement asynchrone. C'est une bonne occasion de parcourir exactement ce qui se passe dans une exécution asynchrone ordinaire :

- Tout d'abord, `main` appelle `cheapo_request` , qui renvoie le futur `A` de son résultat final. `main` Passe ensuite ce futur à `async_std::block_on` , qui l'interroge.
- L' interrogation future `A` permet au corps de `cheapo_request` commencer l'exécution. Il appelle `TcpStream::connect` pour obtenir un futur `B` d'un socket et attend ensuite cela. Plus précisément, puisque `TcpStream::connect` pourrait rencontrer une erreur, `B` est un futur de a `Result<TcpStream, std::io::Error>` .

- Future `B` est interrogé par le `await`. La connexion réseau n'étant pas encore établie, `B.poll` renvoie `Poll::Pending`, mais s'arrange pour réveiller la tâche appelante une fois que le socket est prêt.
- Puisque le futur `B` n'était pas prêt, `A.poll` retourne `Poll::Pending` à son propre appelant, `block_on`.
- Puisqu'il `block_on` n'a rien de mieux à faire, il s'endort. Le fil entier est maintenant bloqué.
- Lorsque `B` la connexion de est prête à être utilisée, elle réveille la tâche qui l'a interrogée. Cela `block_on` se transforme en action et tente à nouveau d'interroger l'avenir `A`.
- L' `A` interrogation `cheapo_request` reprend dans son premier `await`, où elle interroge `B` à nouveau.
- Cette fois, `B` est prêt : la création du socket est terminée, il revient donc `Poll::Ready(Ok(socket))` à `A.poll`.
- L'appel asynchrone à `TcpStream::connect` est maintenant terminé. La valeur de l' `TcpStream::connect(...).await` expression est donc `Ok(socket)`.
- L'exécution du `cheapo_request` corps de se déroule normalement, en créant la chaîne de requête à l'aide de la `format!` macro et en la transmettant à `socket.write_all`.
- Puisque `socket.write_all` est une fonction asynchrone, elle renvoie un futur `C` de son résultat, qui `cheapo_request` attend dûment.

Le reste de l'histoire est similaire. Dans l'exécution illustrée à la Figure 20-2, le futur de `socket.read_to_string` est interrogé quatre fois avant d'être prêt ; chacun de ces réveils lit *certaines* données du socket, mais `read_to_string` est spécifié pour lire jusqu'à la fin de l'entrée, et cela prend plusieurs opérations.

Cela ne semble pas trop difficile d'écrire simplement une boucle qui appelle `poll` encore et encore. Mais ce qui `async_std::task::block_on` est précieux, c'est qu'il sait comment s'endormir jusqu'à ce que l'avenir vaille la peine d'être interrogé à nouveau, plutôt que de gaspiller le temps de votre processeur et la durée de vie de votre batterie à faire des milliards d' `poll` appels infructueux. Les futurs renvoyés par les fonctions d'E/S de base aiment `connect` et `read_to_string` conservent le réveil fourni par le `Context` passé à `poll` et l'invoquent lorsqu'il `block_on` doit se réveiller et essayer à nouveau d'interroger. Nous montrerons exactement comment cela fonctionne en implémentant une version simple de `block_on` nous-mêmes dans « Primitive Futures and Executors : When Is a Future Worth Polling Again ? » .

Comme la version originale et synchrone que nous avons présentée précédemment, cette version asynchrone de `cheapo_request` passe presque tout son temps à attendre la fin des opérations. Si l'axe du temps était dessiné à l'échelle, le diagramme serait presque entièrement gris foncé, avec de minuscules éclats de calcul se produisant lorsque le programme se réveille.

C'est beaucoup de détails. Heureusement, vous pouvez généralement penser en termes de chronologie supérieure simplifiée : certains appels de fonction sont synchronisés, d'autres sont asynchrones et nécessitent un `await`, mais ce ne sont que des appels de fonction. Le succès du support asynchrone de Rust dépend du fait d'aider les programmeurs à travailler avec la vue simplifiée dans la pratique, sans être distraits par les allers-retours de l'implémentation.

Génération de tâches asynchrones

La `async_std::task::block_on` fonctionbloque jusqu'à ce que la valeur d'un contrat à terme soit prête. Mais bloquer complètement un thread sur un seul futur n'est pas mieux qu'un appel synchrone : le but de ce chapitre est de faire en sorte que le thread *fasse d'autres travaux* pendant qu'il attend.

Pour ça, vous pouvez utiliser `async_std::task::spawn_local`. Cette fonctionprend un futur et l'ajoute à un pool qui `block_on` essaiera d'interroger chaque fois que le futur sur lequel il bloque n'est pas prêt. Donc, si vous passez un tas de contrats à terme `spawn_local` et que vous postulez ensuite `block_on` à un contrat à terme de votre résultat final, chaque contrat à `block_on` terme généré sera interrogé chaque fois qu'il est capable de progresser, en exécutant l'ensemble du pool simultanément jusqu'à ce que votre résultat soit prêt.

Au moment d'écrire ces lignes, `spawn_local` n'est disponible `async-std` que si vous activez la `unstable` fonctionnalité de cette caisse. Pour ce faire, vous devrez vous référer à `async-std` dans votre *Cargo.toml* avec une ligne comme celle-ci :

```
async-std = { version = "1", fonctionnalités = ["unstable"] }
```

La `spawn_local` fonction est un analogue asynchrone de la `std::thread::spawn` fonction de la bibliothèque standardpour démarrer les threads :

- `std::thread::spawn(c)` prend une fermeture `c` et démarre un thread qui l'exécute, renvoyant une méthode `std::thread::JoinHandle` dont la `join` méthode attend que le thread se termine et renvoie tout ce qui `c` est renvoyé.
- `async_std::task::spawn_local(f)` prend le futur `f` et l'ajoute au pool pour être interrogé lorsque le thread actuel appelle `block_on` . `spawn_local` renvoie son propre `async_std::task::JoinHandle` type, lui-même un futur que vous pouvez attendre pour récupérer `f` la valeur finale de .

Par exemple, supposons que nous souhaitions effectuer simultanément tout un ensemble de requêtes HTTP. Voici une première tentative :

```
pub async fn many_requests(requests: Vec<(String, u16, String)>)
                           -> Vec<std:: io:: Result<String>>
{
    use async_std::task;

    let mut handles = vec![];
    for (host, port, path) in requests {
        handles.push(task::spawn_local(cheapo_request(&host, port, &path)))
    }

    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }

    results
}
```

Cette fonction appelle `cheapo_request` chaque élément de `requests` , en passant le futur de chaque appel à `spawn_local` . Il collecte les `JoinHandle` s résultants dans un vecteur puis attend chacun d'eux. C'est bien d'attendre les descripteurs de jointure dans n'importe quel ordre : puisque les requêtes sont déjà générées, leur avenir sera interrogé au besoin chaque fois que ce thread appelle `block_on` et n'a rien de mieux à faire. Toutes les requêtes seront exécutées simultanément. Une fois qu'ils sont terminés, `many_requests` renvoie les résultats à son appelant.

Le code précédent est presque correct, mais le vérificateur d'emprunt de Rust s'inquiète de la durée de vie du `cheapo_request` futur de :

```
error: `host` does not live long enough

    handles.push(task::spawn_local(cheapo_request(&host, port, &path)));
```

```
                                          ---------------^^^^^--------------
                                          |              |
                                          |              borrowed value does not
                                          |              live long enough
                          argument requires that `host` is borrowed for `'static
    }
    - `host` dropped here while still borrowed
```

Il y a une erreur similaire pour `path` aussi.

Naturellement, si nous passons des références à une fonction asynchrone, le futur qu'elle renvoie doit contenir ces références, de sorte que le futur ne peut pas survivre en toute sécurité aux valeurs qu'elles empruntent. Il s'agit de la même restriction qui s'applique à toute valeur contenant des références.

Le problème est que `spawn_local` vous ne pouvez pas être sûr que vous attendrez la fin de la tâche avant `host` et que vous serez `path` abandonné. En fait, `spawn_local` n'accepte que les contrats à terme dont la durée de vie est `'static` , car vous pouvez simplement ignorer le `JoinHandle` retour et laisser la tâche continuer à s'exécuter pour le reste de l'exécution du programme. Ce n'est pas propre aux tâches asynchrones : vous obtiendrez une erreur similaire si vous essayez d'utiliser `std::thread::spawn` pour démarrer un thread dont la fermeture capture les références aux variables locales.

Une façon de résoudre ce problème consiste à créer une autre fonction asynchrone qui prend des versions propriétaires des arguments :

```
async fn cheapo_owning_request(host: String, port: u16, path: String)
                                    -> std:: io::Result<String> {
    cheapo_request(&host, port, &path).await
}
```

Cette fonction prend `String` s au lieu de `&str` références, donc son futur possède les chaînes `host` et lui-même, et sa durée de vie est . Le vérificateur d'emprunt peut voir qu'il attend immédiatement le futur de , et par conséquent, si ce futur est interrogé, les variables et qu'il emprunte doivent toujours être présentes. Tout est
bien. `path 'static cheapo_request host path`

En utilisant `cheapo_owning_request` , vous pouvez générer toutes vos requêtes comme suit :

```
    for (host, port, path) in requests {
        handles.push(task::spawn_local(cheapo_owning_request(host, port, path)))
    }
```

Vous pouvez appeler `many_requests` depuis votre fonction synchrone `main`, avec `block_on` :

```
    let requests = vec![
        ("example.com".to_string(),      80, "/".to_string()),
        ("www.red-bean.com".to_string(), 80, "/".to_string()),
        ("en.wikipedia.org".to_string(), 80, "/".to_string()),
    ];

    let results = async_std:: task::block_on(many_requests(requests));
    for result in results {
        match result {
            Ok(response) => println!("{}", response),
            Err(err) => eprintln!("error: {}", err),
        }
    }
```

Ce code exécute les trois requêtes simultanément à partir de l'appel à `block_on`. Chacun progresse au fur et à mesure que l'opportunité se présente tandis que les autres sont bloqués, tous sur le fil appelant. La figure 20-3 montre une exécution possible des trois appels à `cheapo_request`.

(Nous vous encourageons à essayer d'exécuter ce code vous-même, avec des `eprintln!` appels ajoutés en haut `cheapo_request` et après chaque `await` expression afin que vous puissiez voir comment les appels s'entrelacent différemment d'une exécution à l'autre.)

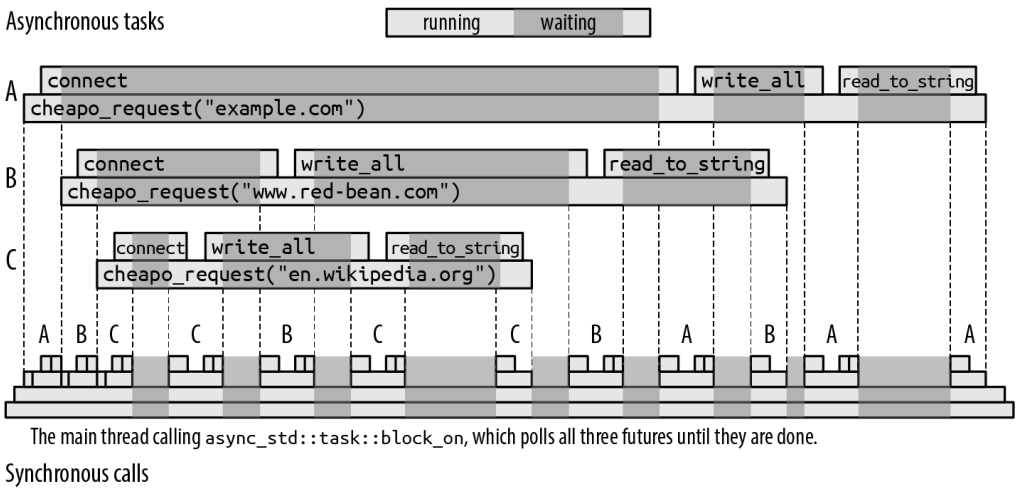

Illustration 20-3. Exécution de trois tâches asynchrones sur un seul thread

L'appel à `many_requests` (non illustré, pour des raisons de simplicité) a engendré trois tâches asynchrones, que nous avons étiquetées `A`, `B` et `C`. `block_on` commence par interroger `A`, qui commence à se connecter à `example.com`. Dès que cela revient `Poll::Pending`, `block_on` tourne son attention vers la prochaine tâche générée, interrogeant future `B`, et éventuellement `C`, qui commencent chacune à se connecter à leurs serveurs respectifs.

Lorsque tous les futurs pollables sont revenus `Poll::Pending`, `block_on` s'endort jusqu'à ce que l'un des `TcpStream::connect` futurs indique que sa tâche vaut la peine d'être interrogée à nouveau.

Dans cette exécution, le serveur `en.wikipedia.org` répond plus rapidement que les autres, de sorte que la tâche se termine en premier. Lorsqu'une tâche générée est terminée, elle enregistre sa valeur dans son `JoinHandle` et la marque comme prête, afin qu'elle `many_requests` puisse continuer lorsqu'elle l'attend. Finalement, les autres appels à `cheapo_request` réussiront ou renverront une erreur, et lui- `many_requests` même peut revenir. Enfin, `main` reçoit le vecteur de résultats de `block_on`.

Toute cette exécution se déroule sur un fil unique, les trois appels `cheapo_request` s'entrelaçant les uns aux autres par des interrogations successives de leurs devenirs. Un appel asynchrone offre l'apparence d'un seul appel de fonction exécuté jusqu'à son terme, mais cet appel asynchrone est réalisé par une série d'appels synchrones à la `poll` méthode du futur. Chaque `poll` appel individuel revient rapidement, produisant le thread afin qu'un autre appel asynchrone puisse prendre son tour.

Nous avons finalement atteint l'objectif que nous avions défini au début du chapitre : laisser un thread s'occuper d'autres tâches en attendant la fin des E/S afin que les ressources du thread ne soient pas occupées à ne rien faire. Mieux encore, cet objectif a été atteint avec un code qui ressemble beaucoup au code Rust ordinaire : certaines des fonctions sont marquées `async`, certains des appels de fonction sont suivis de `.await`, et nous utilisons des fonctions de `async_std` au lieu de `std`, mais sinon, c'est du code Rust ordinaire.

Une différence importante à garder à l'esprit entre les tâches asynchrones et les threads est que le passage d'une tâche asynchrone à une autre ne se produit qu'au niveau `await` des expressions, lorsque le futur attendu renvoie `Poll::Pending`. Cela signifie que si vous mettez un calcul de longue durée dans `cheapo_request`, aucune des autres tâches

que vous avez transmises `spawn_local` n'aura la chance de s'exécuter jusqu'à ce qu'elle soit terminée. Avec les threads, ce problème ne se pose pas : le système d'exploitation peut suspendre n'importe quel thread à tout moment et définir des temporisateurs pour s'assurer qu'aucun thread ne monopolise le processeur. Le code asynchrone dépend de la coopération volontaire des futurs partageant le fil. Si vous avez besoin de faire coexister des calculs de longue durée avec du code asynchrone, "Calculs de longue durée : yield_now et spawn_blocking"plus loin dans ce chapitre décrit certaines options.

Blocs asynchrones

en outreaux fonctions asynchrones, Rust prend également en charge *les blocs asynchrones* . Alors qu'une instruction de bloc ordinaire renvoie la valeur de sa dernière expression, un bloc asynchrone renvoie *un futur de* la valeur de sa dernière expression. Vous pouvez utiliser des `await` expressions dans un bloc asynchrone.

Un bloc asynchrone ressemble à une instruction de bloc ordinaire, précédée du `async` mot clé :

```
let serve_one = async {
    use async_std::net;

    // Listen for connections, and accept one.
    let listener = net:: TcpListener::bind("localhost:8087").await?;
    let (mut socket, _addr) = listener.accept().await?;

    // Talk to client on `socket`.
    ...
};
```

Cela s'initialise `serve_one` avec un futur qui, lorsqu'il est interrogé, écoute et gère une seule connexion TCP. Le corps du bloc ne commence pas son exécution tant qu'il n'a pas `serve_one` été interrogé, tout comme un appel de fonction asynchrone ne commence pas son exécution tant que son avenir n'est pas interrogé.

Si vous appliquez l' `?` opérateur à une erreur dans un bloc asynchrone, il revient simplement du bloc, pas de la fonction environnante. Par exemple, si l'appel précédent `bind` renvoie une erreur, l' `?` opérateur la renvoie comme `serve_one` valeur finale de . De même, `return` les expressions reviennent du bloc asynchrone, pas de la fonction englobante.

Si un bloc asynchrone fait référence à des variables définies dans le code environnant, son futur capture leurs valeurs, tout comme le ferait une fermeture. Et tout comme les `move` fermetures (voir "Closures That Steal"), vous pouvez commencer le blocavec `async move` pour s'approprier les valeurs capturées, plutôt que de se contenter d'y faire référence.

Les blocs asynchrones offrent un moyen concis de séparer une section de code que vous souhaitez exécuter de manière asynchrone. Par exemple, dans la section précédente, il `spawn_local` fallait un `'static` futur, nous avons donc défini la `cheapo_owning_request` fonction wrapper pour nous donner un futur qui s'approprie ses arguments. Vous pouvez obtenir le même effet sans la distraction d'une fonction wrapper simplement en appelant `cheapo_request` depuis un bloc asynchrone :

```
pub async fn many_requests(requests: Vec<(String, u16, String)>)
                           -> Vec<std:: io:: Result<String>>
{
    use async_std::task;

    let mut handles = vec![];
    for (host, port, path) in requests {
        handles.push(task::spawn_local(async move {
            cheapo_request(&host, port, &path).await
        }));
    }
    ...
}
```

Puisqu'il s'agit d'un `async move` bloc, son avenir s'approprie les `String` valeurs `host` et `path` , exactement comme le `move` ferait une fermeture. Il passe ensuite les références à `cheapo_request` . Le vérificateur d'emprunt peut voir que l'expression du bloc `await` s'approprie `cheapo_request` le futur de , de sorte que les références à `host` et `path` ne peuvent pas survivre aux variables capturées qu'elles empruntent. Le bloc async accomplit la même chose que `cheapo_owning_request` , mais avec moins de passe-partout.

Un bord rugueux que vous pouvez rencontrer est qu'il n'y a pas de syntaxe pour spécifier le type de retour d'un bloc asynchrone, analogue aux `-> T` arguments suivants d'une fonction asynchrone. Cela peut poser des problèmes lors de l'utilisation de l' `?` opérateur :

```
let input = async_std:: io:: stdin();
let future = async {
    let mut line = String::new();
```

```
        // This returns `std::io::Result<usize>`.
        input.read_line(&mut line).await?;

        println!("Read line: {}", line);

        Ok(())
    };
```

Cela échoue avec l'erreur suivante :

```
error: type annotations needed
   |
48 |     let future = async {
   |         ------ consider giving `future` a type
...
60 |         Ok(())
   |         ^^ cannot infer type for type parameter `E` declared
   |            on the enum `Result`
```

Rust ne peut pas dire quel doit être le type de retour du bloc asynchrone. La `read_line` méthode renvoie `Result<(), std::io::Error>` , mais comme l' `?` opérateur utilise le `From` trait pour convertir le type d'erreur en question en tout ce que la situation exige, le type de retour du bloc asynchrone peut être `Result<(), E>` pour n'importe quel type `E` qui implémente `From<std::io::Error>` .

Les futures versions de Rust ajouteront probablement une syntaxe pour indiquer `async` le type de retour d'un bloc. Pour l'instant, vous pouvez contourner le problème en épelant le type de final du bloc `Ok` :

```
    let future = async {
        ...
        Ok:: <(), std:: io::Error>(())
    };
```

Puisqu'il `Result` s'agit d'un type générique qui attend les types de succès et d'erreur comme paramètres, nous pouvons spécifier ces paramètres de type lors de l'utilisation de `Ok` ou `Err` comme indiqué ici.

Création de fonctions asynchrones à partir de blocs asynchrones

Asynchroneles blocs nous donnent un autre moyen d'obtenir le même effet qu'une fonction asynchrone, avec un peu plus de flexibilité. Par exemple, nous pourrions écrire notre `cheapo_request` exemple sous la

forme d'une fonction synchrone ordinaire qui renvoie le futur d'un bloc asynchrone :

```
use std:: io;
use std:: future::Future;

fn cheapo_request<'a>(host: &'a str, port: u16, path: &'a str)
    -> impl Future<Output = io::Result<String>> + 'a
{
    async move {
        ... function body ...
    }
}
```

Lorsque vous appelez cette version de la fonction, elle renvoie immédiatement le futur de la valeur du bloc asynchrone. Cela capture les arguments de la fonction et se comporte exactement comme le futur que la fonction asynchrone aurait renvoyé. Puisque nous n'utilisons pas la `async fn` syntaxe, nous devons écrire le `impl Future` dans le type de retour, mais en ce qui concerne les appelants, ces deux définitions sont des implémentations interchangeables de la même signature de fonction.

Cette deuxième approche peut être utile lorsque vous souhaitez effectuer des calculs immédiatement lorsque la fonction est appelée, avant de créer le futur de son résultat. Par exemple, une autre façon de réconcilier serait d'en faire une fonction synchrone renvoyant un `cheapo_request` futur qui capture des copies entièrement possédées de ses arguments : `spawn_local 'static`

```
fn cheapo_request(host: &str, port: u16, path: &str)
    -> impl Future<Output = io::Result<String>> + 'static
{
    let host = host.to_string();
    let path = path.to_string();

    async move {
        ... use &*host, port, and path ...
    }
}
```

Cette version permet au bloc asynchrone de capturer `host` et en tant que valeurs `path` détenues , pas de références. Étant donné que le futur possède toutes les données dont il a besoin pour s'exécuter, il est valide pour la durée de vie. (Nous avons précisé dans la signature montrée plus tôt, mais c'est la valeur par défaut pour les types de retour, donc l'omettre

n'aurait aucun effet `String &str 'static + 'static 'static -> impl` .)

Depuis cette version des `cheapo_request` retours à terme qui sont `'static` , on peut les passer directement à `spawn_local` :

```
let join_handle = async_std:: task::spawn_local(
    cheapo_request("areweasyncyet.rs", 80, "/")
);

... other work ...

let response = join_handle.await?;
```

Génération de tâches asynchrones sur un pool de threads

Les exemplesnous avons montré jusqu'à présent qu'ils passent presque tout leur temps à attendre des E/S, mais certaines charges de travail sont davantage un mélange de travail de processeur et de blocage. Lorsque vous avez suffisamment de calculs pour faire qu'un seul processeur ne puisse pas suivre, vous pouvez utiliser `async_std::task::spawn` pour générer un futur sur un pool de threads de travail dédiés à l'interrogation des futurs prêts à progresser.

`async_std::task::spawn` s'utilise comme `async_std::task::spawn_local` :

```
use async_std::task;

let mut handles = vec![];
for (host, port, path) in requests {
    handles.push(task::spawn(async move {
        cheapo_request(&host, port, &path).await
    }));
}
...
```

Comme `spawn_local` , `spawn` renvoie une `JoinHandle` valeur que vous pouvez attendre pour obtenir la valeur finale du futur. Mais contrairement à `spawn_local` , le futur n'a pas besoin d'attendre que vous appeliez `block_on` avant d'être interrogé. Dès que l'un des threads du pool de threads est libre, il essaie de l'interroger.

En pratique, `spawn` est plus largement utilisé que `spawn_local`, simplement parce que les gens aiment savoir que leur charge de travail, quelle que soit sa combinaison de calcul et de blocage, est équilibrée entre les ressources de la machine.

Une chose à garder à l'esprit lors de l'utilisation `spawn` est que le pool de threads essaie de rester occupé, de sorte que votre avenir est interrogé par le thread qui l'aborde en premier. Un appel asynchrone peut commencer son exécution sur un thread, se bloquer sur une `await` expression et reprendre dans un autre thread. Ainsi, bien qu'il soit raisonnable de considérer un appel de fonction asynchrone comme une exécution de code unique et connectée (en effet, le but des fonctions et `await` des expressions asynchrones est de vous encourager à y penser de cette façon), l'appel peut en fait être effectué par beaucoup de fils différents.

Si vous utilisez le stockage local de thread, il peut être surprenant de voir les données que vous y placez avant une `await` expression remplacées par quelque chose de complètement différent par la suite, car votre tâche est maintenant interrogée par un thread différent du pool. Si cela pose problème, vous devez plutôt utiliser *le stockage local de la tâche* ; voir la `async-std` documentation de la caisse pour la `task_local!` macropour plus de détails.

Mais votre futur outil envoie-t-il ?

Làest une restriction `spawn` impose qui `spawn_local` ne le fait pas. Étant donné que le futur est envoyé à un autre thread pour s'exécuter, le futur doit implémenter le `Send` trait de marqueur. Nous avons présenté `Send` dans « Thread Safety : Send and Sync » . Un futur `Send` n'existe que si toutes les valeurs qu'il contient sont `Send` : tous les arguments de la fonction, les variables locales et même les valeurs temporaires anonymes doivent pouvoir être déplacés en toute sécurité vers un autre thread.

Comme précédemment, cette exigence n'est pas unique aux tâches asynchrones : vous obtiendrez une erreur similaire si vous essayez d'utiliser `std::thread::spawn` pour démarrer un thread dont la fermeture capture des non `Send` -valeurs. La différence est que, alors que la fermeture transmise à `std::thread::spawn` reste sur le thread qui a été créé pour l'exécuter, un futur généré sur un pool de threads peut passer d'un thread à un autre à tout moment.

Cette restriction est facile à trébucher par accident. Par exemple, le code suivant semble assez innocent :

```
use async_std:: task;
use std:: rc::Rc;

async fn reluctant() -> String {
    let string = Rc::new("ref-counted string".to_string());

    some_asynchronous_thing().await;

    format!("Your splendid string: {}", string)
}

task::spawn(reluctant());
```

Le futur d'une fonction asynchrone doit contenir suffisamment d'informations pour que la fonction continue à partir d'une `await` expression. Dans ce cas, `reluctant` le futur de doit être utilisé `string` après le `await` , donc le futur contiendra, au moins parfois, une `Rc<String>` valeur. Étant donné que `Rc` les pointeurs ne peuvent pas être partagés en toute sécurité entre les threads, le futur lui-même ne peut pas être `Send` . Et puisque `spawn` n'accepte que les futurs qui sont `Send` , les objets Rust :

```
error: future cannot be sent between threads safely
    |
17  |     task::spawn(reluctant());
    |     ^^^^^^^^^^^^ future returned by `reluctant` is not `Send`
    |

    |
127 | T: Future + Send + 'static,
    |             ---- required by this bound in `async_std::task::spawn`
    |
    = help: within `impl Future`, the trait `Send` is not implemented
            for `Rc<String>`
note: future is not `Send` as this value is used across an await
    |
10  |         let string = Rc::new("ref-counted string".to_string());
    |             ------ has type `Rc<String>` which is not `Send`
11  |
12  |         some_asynchronous_thing().await;
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                    await occurs here, with `string` maybe used later
...
15  |     }
    |     - `string` is later dropped here
```

Ce message d'erreur est long, mais il contient de nombreux détails utiles :

- Il explique pourquoi l'avenir doit être `Send` : l' `task::spawn` exige.

- Il explique quelle valeur n'est pas `Send` : la variable locale `string`, dont le type est `Rc<String>`.
- Il explique pourquoi `string` affecte l'avenir: il est dans la portée à travers l'indiqué `await`.

Il existe deux façons de résoudre ce problème. L'une consiste à re-streindre la portée de la non- `Send` valeur afin qu'elle ne couvre aucune `await` expression et n'ait donc pas besoin d'être enregistrée dans le futur de la fonction :

```
async fn reluctant() -> String {
    let return_value = {
        let string = Rc::new("ref-counted string".to_string());
        format!("Your splendid string: {}", string)
        // The `Rc<String>` goes out of scope here...
    };

    // ... and thus is not around when we suspend here.
    some_asynchronous_thing().await;

    return_value
}
```

Une autre solution consiste simplement à utiliser à la `std::sync::Arc` place de `Rc`. `Arc` utilise des mises à jour atomiques pour gérer son nombre de références, ce qui le rend un peu plus lent, mais `Arc` les pointeurs sont `Send`.

Bien que vous finirez par apprendre à reconnaître et à éviter les non `Send` -types, ils peuvent être un peu surprenants au début. (Au moins, vos auteurs ont souvent été surpris.) Par exemple, le code Rust plus ancien utilise parfois des types de résultats génériques comme celui-ci :

```
// Not recommended!
type GenericError = Box<dyn std:: error::Error>;
type GenericResult<T> = Result<T, GenericError>;
```

Ce `GenericError` type utilise un objet de trait encadré pour contenir une valeur de n'importe quel type qui implémente `std::error::Error`. Mais cela n'impose aucune autre restriction : si quelqu'un avait un non `Send` -type qui implémentait `Error`, il pourrait convertir une valeur encadrée de ce type en un `GenericError`. En raison de cette possibilité, `GenericError` n'est pas `Send`, et le code suivant ne fonctionnera pas :

```
fn some_fallible_thing() ->GenericResult<i32> {
    ...
}

// This function's future is not `Send`...
async fn unfortunate() {
    // ... because this call's value ...
    match some_fallible_thing() {
        Err(error) => {
            report_error(error);
        }
        Ok(output) => {
            // ... is alive across this await ...
            use_output(output).await;
        }
    }
}

// ... and thus this `spawn` is an error.
async_std:: task::spawn(unfortunate());
```

Comme dans l'exemple précédent, le message d'erreur du compilateur explique ce qui se passe, pointant vers le `Result` type comme coupable. Puisque Rust considère que le résultat de `some_fallible_thing` est présent pour toute la `match` déclaration, y compris l' `await` expression, il détermine que le futur de `unfortunate` n'est pas `Send` . Cette erreur est trop prudente de la part de Rust : bien qu'il soit vrai qu'il `GenericError` n'est pas sûr de l'envoyer à un autre thread, cela `await` ne se produit que lorsque le résultat est `Ok` , donc la valeur d'erreur n'existe jamais réellement lorsque nous attendons `use_output` le futur de .

La solution idéale consiste à utiliser des types d'erreurs génériques plus stricts comme ceux que nous avons suggérés dans "Travailler avec plusieurs types d'erreurs" :

```
type GenericError = Box<dyn std:: error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

Cet objet de trait nécessite explicitement que le type d'erreur sous-jacent soit implémenté `Send` , et tout va bien.

Si votre avenir ne l'est pas `Send` et que vous ne pouvez pas le faire facilement, vous pouvez toujours l'utiliser `spawn_local` pour l'exécuter sur le thread actuel. Bien sûr, vous devrez vous assurer que le thread appelle `block_on` à un moment donné, pour lui donner une chance de s'exécu-

ter, et vous ne bénéficierez pas de la distribution du travail sur plusieurs processeurs..

Calculs de longue durée : yield_now et spawn_blocking

Pour un avenirpartager agréablement son fil avec d'autres tâches, sa `poll` méthodedoit toujours revenir le plus rapidement possible. Mais si vous effectuez un long calcul, cela peut prendre beaucoup de temps pour atteindre le prochain `await`, ce qui oblige les autres tâches asynchrones à attendre plus longtemps que vous ne le souhaiteriez pour leur tour sur le thread.

Une façon d'éviter cela est simplement de faire `await` quelque chose de temps en temps. La `async_std::task::yield_now` fonctionrenvoie un futur simple conçu pour cela :

```
while computation_not_done() {
    ... do one medium-sized step of computation ...
    async_std:: task::yield_now().await;
}
```

La première fois que l' `yield_now` avenir est interrogé, il revient `Poll::Pending`, mais indique qu'il vaut la peine d'être interrogé à nouveau bientôt. L'effet est que votre appel asynchrone abandonne le thread et que d'autres tâches ont une chance de s'exécuter, mais votre appel aura bientôt un autre tour. La seconde fois `yield_now` que le futur est interrogé, il renvoie `Poll::Ready(())` et votre fonction asynchrone peut reprendre son exécution.

Cette approche n'est cependant pas toujours réalisable. Si vous utilisez une caisse externe pour effectuer le calcul de longue durée ou pour appeler C ou C++, il peut ne pas être pratique de modifier ce code pour qu'il soit plus convivial pour l'asynchronisme. Ou il peut être difficile de s'assurer que chaque chemin à travers le calcul est sûr d'atteindre le `await` de temps en temps.

Pour des cas comme celui-ci, vous pouvez utiliser `async_std::task::spawn_blocking`. Cette fonction prend une fermeture, la lance sur son propre thread et renvoie un futur de sa valeur de retour. Le code asynchrone peut attendre ce futur, cédant son fil à d'autres tâches jusqu'à ce que le calcul soit prêt. En mettant le travail acharné sur un thread séparé, vous pouvez laisser le système d'exploitation s'occuper de lui faire partager le processeur de manière agréable.

Par exemple, supposons que nous devions vérifier les mots de passe fournis par les utilisateurs par rapport aux versions hachées que nous avons stockées dans notre base de données d'authentification. Pour des raisons de sécurité, la vérification d'un mot de passe doit nécessiter beaucoup de calculs, de sorte que même si les attaquants obtiennent une copie de notre base de données, ils ne peuvent pas simplement essayer des milliards de mots de passe possibles pour voir s'ils correspondent. La `argonautica` caisse fournit une fonction de hachage conçue spécifiquement pour stocker les mots de passe : un `argonautica` hachage correctement généré prend une fraction de seconde significative à vérifier. Nous pouvons utiliser `argonautica` (version `0.2`) dans notre application asynchrone comme ceci :

```
async fn verify_password(password: &str, hash: &str, key: &str)
                         -> Result<bool, argonautica::Error>
{
    // Make copies of the arguments, so the closure can be 'static.
    let password = password.to_string();
    let hash = hash.to_string();
    let key = key.to_string();

    async_std:: task:: spawn_blocking(move || {
        argonautica:: Verifier::default()
            .with_hash(hash)
            .with_password(password)
            .with_secret_key(key)
            .verify()
    }).await
}
```

Cela renvoie `Ok(true)` si `password` correspond à `hash`, étant donné `key`, une clé pour la base de données dans son ensemble. En effectuant la vérification dans la fermeture transmise à `spawn_blocking`, nous poussons le calcul coûteux sur son propre thread, en veillant à ce qu'il n'affecte pas notre réactivité aux demandes des autres utilisateurs.

Comparaison de conceptions asynchrones

Dans de nombreuxl'approche de Rust en matière de programmation asynchrone ressemble à celle adoptée par d'autres langages. Par exemple, JavaScript, C# et Rust ont tous des fonctions asynchrones avec des `await` expressions. Et tous ces langages ont des valeurs qui représentent des calculs incomplets : Rust les appelle « futurs », JavaScript les appelle « promesses » et C# appelleles «tâches», mais elles représentent toutes une valeur que vous devrez peut-être attendre.

L'utilisation des sondages par Rust, cependant, est inhabituel. En Javascriptet C#, une fonction asynchrone commence à s'exécuter dès qu'elle est appelée, et il existe une boucle d'événements globale intégrée à la bibliothèque système qui reprend les appels de fonction asynchrone suspendus lorsque les valeurs qu'ils attendaient deviennent disponibles. Dans Rust, cependant, un appel asynchrone ne fait rien jusqu'à ce que vous le passiez à une fonction comme `block_on`, `spawn` ou `spawn_local` qui l'interrogera et conduira le travail à son terme. Ces fonctions, appelées *exécuteurs*, jouent le rôle que d'autres langages couvrent avec une boucle d'événements globale.

Parce que Rust vous oblige, le programmeur, à choisir un exécuteur pour interroger votre avenir, Rust n'a pas besoin d'une boucle d'événements globaleintégré au système. La `async-std` caisse offre les fonctions d'exécuteur que nous avons utilisées dans ce chapitre jusqu'ici, mais la `tokio` caisse, que nous utiliserons plus tard dans ce chapitre, définit son propre ensemble de fonctions d'exécution similaires. Et vers la fin de ce chapitre, nous implémenterons notre propre exécuteur. Vous pouvez utiliser les trois dans le même programme.

Un vrai client HTTP asynchrone

Nous serions négligents si nous ne montrions pas un exemple d'utilisation d'une caisse de client HTTP asynchrone appropriée, car c'est si facile, et il y a plusieurs bonnes caisses parmi lesquelles choisir, y compris `reqwest` et `surf`.

Voici une réécriture de `many_requests`, encore plus simple que celle basée sur `cheapo_request`, qui `surf` permet d'exécuter simultanément une série de requêtes. Vous aurez besoin de ces dépendances dans votre fichier *Cargo.toml* :

```
[dépendances]
async-std = "1.7"
surf = "1.0"
```

Ensuite, nous pouvons définir `many_requests` comme suit :

```
pub async fn many_requests(urls: &[String])
                           -> Vec<Result<String, surf:: Exception>>
{
    let client = surf:: Client::new();

    let mut handles = vec![];
```

```
    for url in urls {
        let request = client.get(&url).recv_string();
        handles.push(async_std:: task::spawn(request));
    }

    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }

    results
}

fn main() {
    let requests = &["http://example.com".to_string(),
                     "https://www.red-bean.com".to_string(),
                     "https://en.wikipedia.org/wiki/Main_Page".to_string()];

    let results = async_std:: task::block_on(many_requests(requests));
    for result in results {
        match result {
            Ok(response) => println!("*** {}\n", response),
            Err(err) => eprintln!("error: {}\n", err),
        }
    }
}
```

Utiliser un seul `surf::Client` pour faire toutes nos requêtes nous permet de réutiliser les connexions HTTP si plusieurs d'entre elles sont dirigées vers le même serveur. Et aucun bloc async n'est nécessaire : puisque `recv_string` c'est une méthode asynchrone qui renvoie un `Send + 'static` futur, nous pouvons passer son futur directementà `spawn` .

Un client et un serveur asynchrones

C'est l'heurede prendre les idées clés dont nous avons discuté jusqu'à présent et de les assembler dans un programme de travail. Dans une large mesure, les applications asynchrones ressemblent aux applications multithread ordinaires, mais il existe de nouvelles opportunités de code compact et expressif que vous pouvez rechercher.

L'exemple de cette section est un serveur et un client de chat. Découvrez le code complet . Les vrais systèmes de chat sont compliqués, avec des préoccupations allant de la sécurité et de la reconnexion à la confidentialité et à la modération, mais nous avons réduit le nôtre à un ensemble

austère de fonctionnalités afin de nous concentrer sur quelques points d'intérêt.

En particulier, nous voulons bien gérer *la contre-pression* . Nous entendons par là que si un client a une connexion Internet lente ou abandonne complètement sa connexion, cela ne doit jamais affecter la capacité des autres clients à échanger des messages à leur propre rythme. Et puisqu'un client lent ne devrait pas obliger le serveur à dépenser une mémoire illimitée pour conserver son arriéré de messages sans cesse croissant, notre serveur devrait abandonner les messages pour les clients qui ne peuvent pas suivre, mais les avertir que leur flux est incomplet. (Un vrai serveur de chat enregistrerait les messages sur le disque et permettrait aux clients de récupérer ceux qu'ils ont manqués, mais nous avons laissé cela de côté.)

On démarre le projet avec la commande `cargo new --lib async-chat` et on met le texte suivant dans *async-chat/Cargo.toml* :

```
[forfait]
nom = "chat asynchrone"
version = "0.1.0"
auteurs = ["Vous <vous@exemple.com>"]
édition = "2021"

[dépendances]
async-std = { version = "1.7", fonctionnalités = ["unstable"] }
tokio = { version = "1.0", fonctionnalités = ["sync"] }
serde = { version = "1.0", fonctionnalités = ["dériver", "rc"] }
serde_json = "1.0"
```

Nous dépendons de quatre caisses :

- La `async-std` caisse est la collection de primitives et d'utilitaires d'E/S asynchrones que nous avons utilisés tout au long de ce chapitre.
- La `tokio` caisseest une autre collection de primitives asynchrones comme `async-std` , l'une des plus anciennes et des plus matures. Il est largement utilisé et sa conception et sa mise en œuvre respectent des normes élevées, mais son utilisation nécessite un peu plus de soin que `async-std` .
  `tokio` est une grande caisse, mais nous n'en avons besoin que d'un seul composant, de sorte que le `features = ["sync"]` champ de la ligne de dépendance *Cargo.toml* `tokio` se limite aux pièces dont nous avons besoin, ce qui en fait une dépendance légère.
  Lorsque l'écosystème de la bibliothèque asynchrone était moins mature, les gens évitaient d'utiliser les deux `tokio` et `async-std` dans le

même programme, mais les deux projets ont coopéré pour s'assurer que cela fonctionne, tant que les règles documentées de chaque caisse sont suivies.
- Les caisses `serde` et `serde_json` nous avons vu auparavant, au chapitre 18 . Ceux-ci nous donnent des outils pratiques et efficaces pour générer et analyser JSON, que notre protocole de chat utilise pour représenter les données sur le réseau. Nous voulons utiliser certaines fonctionnalités facultatives de `serde` , nous les sélectionnons donc lorsque nous donnons la dépendance.

La structure entière de notre application de chat, client et serveur, ressemble à ceci :

```
async-chat
├── Cargo.toml
└── src
    ├── lib.rs
    ├── utils.rs
    └── bin
        ├── client.rs
        └── server
            ├── main.rs
            ├── connection.rs
            ├── group.rs
            └── group_table.rs
```

Cette disposition de paquet utilise une fonctionnalité Cargo dont nous avons parlé dans « Le répertoire src/bin » : en plus de la caisse principale de la bibliothèque, *src/lib.rs* , avec son sous-module *src/utils.rs* , il comprend également deux exécutables :

- *src/bin/client.rs* est un fichier exécutable unique pour le client de chat.
- *src/bin/server* est l'exécutable du serveur, réparti sur quatre fichiers : *main.rs* contient la `main` fonction, et il y a trois sous-modules, *connection.rs* , *group.rs* et *group_table.rs* .

Nous présenterons le contenu de chaque fichier source au cours du chapitre, mais une fois qu'ils sont tous en place, si vous tapez `cargo build` dans cet arbre, cela compile la bibliothèque crate puis construit les deux exécutables. Cargo inclut automatiquement la caisse de la bibliothèque en tant que dépendance, ce qui en fait un endroit pratique pour mettre les définitions partagées par le client et le serveur. De même, `cargo check` vérifie l'intégralité de l'arborescence des sources. Pour exécuter l'un ou l'autre des exécutables, vous pouvez utiliser des commandes comme celles-ci :

```
$cargo run --release --bin serveur -- localhost:8088
 $cargo run --release --bin client -- localhost:8088
```

L' `--bin` option indique quel exécutable exécuter et tous les arguments suivant l' `--` option sont transmis à l'exécutable lui-même. Notre client et notre serveur veulent juste connaître l'adresse et le port TCP du serveur.

Types d'erreur et de résultat

`utils` Le module de la caisse de la bibliothèquedéfinit le résultat et l'erreurtypes que nous utiliserons tout au long de l'application. Depuis *src/utils.rs* :

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

pub type ChatError = Box<dyn Error + Send + Sync + 'static>;
pub type ChatResult<T> = Result<T, ChatError>;
```

Il s'agit des types d'erreurs à usage général que nous avons suggérés dans « Utilisation de plusieurs types d'erreurs » . Les `async_std` , `serde_json` et `tokio` crates définissent chacun leurs propres types d'erreur, mais l' `?` opérateur peut les convertir automatiquement en a `ChatError` , en utilisant l'implémentation de la bibliothèque standard du `From` trait qui peut convertir n'importe quel type d'erreur approprié en `Box<dyn Error + Send + Sync + 'static>` . Les limites `Send` et `Sync` garantissent que si une tâche générée sur un autre thread échoue, elle peut signaler l'erreur en toute sécurité au thread principal.

Dans une application réelle, pensez à utiliser la `anyhow` caisse, qui fournit `Error` et `Result` typesimilaires à ceux-ci. La `anyhow` caisse est facile à utiliser et offre des fonctionnalités intéressantes au-delà de ce que notre `ChatError` et `ChatResult` peuvent offrir.

Le protocole

La caisse de la bibliothèquecapture l'intégralité de notre protocole de chatdans ces deux types, définis dans *lib.rs* :

```
use serde:: {Deserialize, Serialize};
use std:: sync::Arc;

pub mod utils;

#[derive(Debug, Deserialize, Serialize, PartialEq)]
```

```
pub enum FromClient {
    Join { group_name: Arc<String> },
    Post {
        group_name: Arc<String>,
        message:Arc<String>,
    },
}

#[derive(Debug, Deserialize, Serialize, PartialEq)]
pub enum FromServer {
    Message {
        group_name: Arc<String>,
        message:Arc<String>,
    },
    Error(String),
}


#[test]
fn test_fromclient_json() {
    use std:: sync::Arc;

    let from_client = FromClient:: Post {
        group_name: Arc:: new("Dogs".to_string()),
        message: Arc::new("Samoyeds rock!".to_string()),
    };

    let json = serde_json::to_string(&from_client).unwrap();
    assert_eq!(json,
               r#"{"Post":{"group_name":"Dogs","message":"Samoyeds rock!"}}

    assert_eq!(serde_json:: from_str::<FromClient>(&json).unwrap(),
               from_client);
}
```

L' `FromClient` énumération représente les paquets qu'un client peut envoyer au serveur : il peut demander à rejoindre un groupe et envoyer des messages à n'importe quel groupe qu'il a rejoint. `FromServer` représente ce que le serveur peut renvoyer : messages postés à un groupe et messages d'erreur. L'utilisation d'une référence comptée `Arc<String>` au lieu d'une plaine `String` aide le serveur à éviter de faire des copies de chaînes lorsqu'il gère des groupes et distribue des messages.

Les `#[derive]` attributs indiquent à la `serde` caisse de générer des implémentations de ses traits `Serialize` et pour et . Cela nous permet d'appeler pour les convertir en valeurs JSON, de les envoyer sur le réseau et enfin d'appeler pour les reconvertir dans leurs formes

Rust. `Deserialize FromClient FromServer serde_json::to_string serde_json::from_str`

Le `test_fromclient_json` test unitaire illustre comment cela est utilisé. Compte tenu de l' `Serialize` implémentation dérivée de `serde`, nous pouvons appeler `serde_json::to_string` pour transformer la `FromClient` valeur donnée en ce JSON :

```
{"Post":{"group_name":"Dogs","message":"Samoyeds rock!"}}
```

Ensuite, l' `Deserialize` implémentation dérivée analyse cela en une `FromClient` valeur équivalente. Notez que les `Arc` pointeurs dans `FromClient` n'ont aucun effet sur le formulaire sérialisé : les chaînes comptées par référence apparaissent directement en tant que valeurs de membre d'objet JSON.

Prendre l'entrée de l'utilisateur : flux asynchrones

Notre conversationLa première responsabilité du client est de lire les commandes de l'utilisateur et d'envoyer les paquets correspondants au serveur. La gestion d'une interface utilisateur appropriée dépasse le cadre de ce chapitre, nous allons donc faire la chose la plus simple qui fonctionne : lire des lignes directement à partir de l'entrée standard. Le code suivant va dans *src/bin/client.rs* :

```
use async_std:: prelude:: *;
use async_chat:: utils:: {self, ChatResult};
use async_std:: io;
use async_std::net;

async fn send_commands(mut to_server: net:: TcpStream) ->ChatResult<()> {
    println!("Commands:\n\
              join GROUP\n\
              post GROUP MESSAGE...\n\
              Type Control-D (on Unix) or Control-Z (on Windows) \
              to close the connection.");

    let mut command_lines = io:: BufReader:: new(io::stdin()).lines();
    while let Some(command_result) = command_lines.next().await {
        let command = command_result?;
        // See the GitHub repo for the definition of `parse_command`.
        let request = match parse_command(&command) {
            Some(request) => request,
            None => continue,
        };
```

```
            utils::send_as_json(&mut to_server, &request).await?;
            to_server.flush().await?;
        }

        Ok(())
    }
```

Cela appelle `async_std::io::stdin` pour obtenir un handle asynchrone sur l'entrée standard du client, l'enveloppe dans un `async_std::io::BufReader` pour le mettre en mémoire tampon, puis appelle `lines` pour traiter l'entrée de l'utilisateur ligne par ligne. Il essaie d'analyser chaque ligne comme une commande correspondant à une `FromClient` valeur et, s'il réussit, envoie cette valeur au serveur. Si l'utilisateur saisit une commande non reconnue, `parse_command` imprime un message d'erreur et renvoie `None` , il `send_commands` peut donc à nouveau faire le tour de la boucle. Si l'utilisateur tape une indication de fin de fichier, le `lines` flux renvoie `None` , et `send_commands` revient. Cela ressemble beaucoup au code que vous écririez dans un programme synchrone ordinaire, sauf qu'il utilise `async_std` les versions des fonctionnalités de la bibliothèque.

La méthode `BufReader` de l' asynchrone `lines` est intéressant. Elle ne peut pas renvoyer d'itérateur, comme le fait la bibliothèque standard : la `Iterator::next` méthode est une fonction synchrone ordinaire, donc l'appel `command_lines.next()` bloquerait le thread jusqu'à ce que la ligne suivante soit prête. Au lieu de cela, `lines` renvoie un *flux* de `Result<String>` valeurs. Un flux est l'analogue asynchrone d'un itérateur : il produit une séquence de valeurs à la demande, de manière asynchrone. Voici la définition du `Stream` trait, du `async_std::stream` module :

```
    trait Stream {
        type Item;

        // For now, read `Pin<&mut Self>` as `&mut Self`.
        fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
            -> Poll<Option<Self::Item>>;
    }
```

Vous pouvez considérer cela comme un hybride des traits `Iterator` et . `Future` Comme un itérateur, a `Stream` a un type associé `Item` et utilise `Option` pour indiquer quand la séquence est terminée. Mais comme un futur, un flux doit être interrogé : pour obtenir l'élément suivant (ou savoir que le flux est terminé), vous devez appeler `poll_next` jusqu'à ce

qu'il renvoie `Poll::Ready`. L'implémentation d'un flux `poll_next` doit toujours revenir rapidement, sans blocage. Et si un flux revient `Poll::Pending`, il doit avertir l'appelant quand cela vaut la peine d'interroger à nouveau via le `Context`.

La `poll_next` méthode est difficile à utiliser directement, mais vous n'aurez généralement pas besoin de le faire. Comme les itérateurs, les flux ont une large collection de méthodes utilitaires telles que `filter` et `map`. Parmi celles-ci se trouve une `next` méthode, qui renvoie un futur du flux suivant `Option<Self::Item>`. Plutôt que d'interroger explicitement le flux, vous pouvez appeler `next` et attendre le futur qu'il renvoie à la place.

L' assemblage de ces éléments `send_commands` consomme le flux de lignes d'entrée en bouclant les valeurs produites par un flux utilisant `next` with `while let` :

```
while let Some(item) = stream.next().await {
    ... use item ...
}
```

(Les futures versions de Rust introduiront probablement une variante asynchrone de la `for` syntaxe de boucle pour consommer des flux, tout comme une boucle ordinaire `for` consomme des `Iterator` valeurs.)

Interroger un flux après sa fin, c'est-à-dire après qu'il est revenu `Poll::Ready(None)` pour indiquer la fin du flux, revient àappeler `next` un itérateur après son retour `None` ou interroger un futur après son retour `Poll::Ready` : le `Stream` trait ne précise pas ce que le flux doit faire, et certains flux peuvent mal se comporter. Comme les futurs et les itérateurs, les flux ont une `fuse` méthode pour s'assurer que ces appels se comportent de manière prévisible, lorsque cela est nécessaire ; voir la documentation pour plus de détails.

Lorsque vous travaillez avec des flux, il est important de ne pas oublier d'utiliser le `async_std` prélude :

```
use async_std:: prelude::*;
```

C'est parce que les méthodes d'utilité pour le `Stream` trait, comme `next`, `map`, `filter`, etc., ne sont en fait pas définies sur `Stream` elles-mêmes. Au lieu de cela, ce sont des méthodes par défaut d'un trait distinct, `StreamExt`, qui est automatiquement implémenté pour tous les `Stream`s :

```
pub trait StreamExt:Stream {
    ... define utility methods as default methods ...
}

impl<T:Stream> StreamExt for T { }
```

Ceci est un exemple du *trait d'extension*modèle que nous avons décrit dans "Traits et types d'autres personnes" . Le `async_std::prelude` moduleapporte les `StreamExt` méthodes dans la portée, donc l'utilisation du prélude garantit que ses méthodes sont visibles dans votre code.

Envoi de paquets

Pour transmettrepaquets sur une socket réseau, notre client et notre serveur utilisent la `send_as_json` fonction `utils` du module de notre caisse de bibliothèque :

```
use async_std:: prelude:: *;
use serde:: Serialize;
use std:: marker::Unpin;

pub async fn send_as_json<S, P>(outbound: &mut S, packet: &P) -> ChatResult
where
    S: async_std:: io:: Write + Unpin,
    P: Serialize,
{
    let mut json = serde_json::to_string(&packet)?;
    json.push('\n');
    outbound.write_all(json.as_bytes()).await?;
    Ok(())
}
```

Cette fonction construit la représentation JSON de `packet` en tant que `String` , ajoute une nouvelle ligne à la fin, puis écrit le tout dans `outbound` .

D'après sa `where` clause, vous pouvez voir que `send_as_json` c'est assez flexible. Le type de paquet à envoyer, `P` , peut être tout ce qui implémente `serde::Serialize` . Le flux de sortie `S` peut être tout ce qui implémente `async_std::io::Write` , la version asynchrone du `std::io::Write` trait pour les flux de sortie. Cela nous suffit pour envoyer `FromClient` et des `FromServer` valeurs sur un fichier `TcpStream` . Garder la définition de `send_as_json` générique garantit qu'il ne dépend pas des détails des types de flux ou de paquets de ma-

nière surprenante : `send_as_json` ne peut utiliser que des méthodes à partir de ces traits.

La `Unpin` contrainteon `S` est nécessaire pour utiliser la `write_all` méthode. Nous aborderons l'épinglage et le désépinglage plus loin dans ce chapitre, mais pour le moment, il devrait suffire d'ajouter des `Unpin` contraintes aux variables de type là où c'est nécessaire ; le compilateur Rust signalera ces cas si vous oubliez.

Plutôt que de sérialiser le paquet directement dans le `outbound` flux, le `send_as_json` sérialise dans un fichier temporaire `String` , puis l'écrit dans `outbound` . Le `serde_json` crate fournit des fonctions pour sérialiser les valeurs directement dans les flux de sortie, mais ces fonctions ne prennent en charge que les flux synchrones. L'écriture dans des flux asynchrones nécessiterait des changements fondamentaux à la fois `serde_json` et dans le `serde` noyau indépendant du format de la caisse, car les traits autour desquels ils sont conçus ont des méthodes synchrones.

Comme pour les flux, de nombreuses méthodes des `async_std` traits d'E/S de sont en fait définies sur des traits d'extension, il est donc important de s'en souvenir `use async_std::prelude::*` chaque fois que vous les utilisez.

Réception de paquets : davantage de flux asynchrones

Pour recevoirpaquets, notre serveur et notre client utiliseront cette fonction du `utils` module pour recevoir `FromClient` et les `FromServer` valeurs d'un socket TCP tamponné asynchrone, un `async_std::io::BufReader<TcpStream>` :

```
use serde:: de::DeserializeOwned;

pub fn receive_as_json<S, P>(inbound: S) -> impl Stream<Item = ChatResult<P
    where S: async_std:: io:: BufRead + Unpin,
          P: DeserializeOwned,
{
    inbound.lines()
        .map(|line_result| -> ChatResult<P> {
            let line = line_result?;
            let parsed = serde_json:: from_str::<P>(&line)?;
            Ok(parsed)
        })
}
```

Comme `send_as_json`, cette fonction est générique dans le flux d'entrée et les types de paquets :

- Le type de flux `S` doit implémenter `async_std::io::BufRead`, l'analogue asynchrone de `std::io::BufRead`, représentant un flux d'octets d'entrée mis en mémoire tampon.
- Le type de paquet `P` doit implémenter `DeserializeOwned`, une variante plus stricte du `serde` trait `Deserialize` de. Pour plus d'efficacité, `Deserialize` peut produire `&str` et des `&[u8]` valeurs qui empruntent leur contenu directement à la mémoire tampon à partir de laquelle elles ont été désérialisées, pour éviter de copier des données. Dans notre cas, cependant, cela ne sert à rien : nous devons renvoyer les valeurs désérialisées à notre appelant, afin qu'elles puissent survivre aux tampons à partir desquels nous les avons analysées. Un type qui implémente `DeserializeOwned` est toujours indépendant du tampon à partir duquel il a été désérialisé.

L'appel `inbound.lines()` donnenous un `Stream` de `std::io::Result<String>` valeurs. Nous utilisons ensuite l'adaptateur du flux `map` pour appliquer une fermeture à chaque élément, en gérant les erreurs et en analysant chaque ligne sous la forme JSON d'une valeur de type `P`. Cela nous donne un flux de `ChatResult<P>` valeurs, que nous renvoyons directement. Le type de retour de la fonction est :

```
impl Stream<Item = ChatResult<P>>
```

Cela indique que nous renvoyons *un* type qui produit une séquence de `ChatResult<P>` valeurs de manière asynchrone, mais notre appelant ne peut pas dire exactement de quel type il s'agit. Étant donné que la fermeture à laquelle nous passons `map` a de toute façon un type anonyme, c'est le type le plus spécifique qui `receive_as_json` pourrait éventuellement être renvoyé.

Notez que ce `receive_as_json` n'est pas, en soi, une fonction asynchrone. C'est une fonction ordinaire qui renvoie une valeur asynchrone, un flux. Comprendre les mécanismes du support asynchrone de Rust plus en profondeur que "juste ajouter `async` et `.await` partout" ouvre le potentiel pour des définitions claires, flexibles et efficaces comme celle-ci qui tirent pleinement parti du langage.

Pour voir comment `receive_as_json` est utilisé, voici la `handle_replies` fonction de notre client de chat de *src/bin/client.rs*, qui reçoit un flux de `FromServer` valeurs du réseau et les imprime pour que l'utilisateur puisse les voir :

```
use async_chat::FromServer;

async fn handle_replies(from_server: net:: TcpStream) -> ChatResult<()> {
    let buffered = io:: BufReader:: new(from_server);
    let mut reply_stream = utils::receive_as_json(buffered);

    while let Some(reply) = reply_stream.next().await {
        match reply? {
            FromServer:: Message { group_name, message } => {
                println!("message posted to {}: {}", group_name, message);
            }
            FromServer::Error(message) => {
                println!("error from server: {}", message);
            }
        }
    }

    Ok(())
}
```

Cette fonction prend une socket recevant des données du serveur, l' `BufReader` entoure d'un (notez bien, la `async_std` version), puis la transmet à pour obtenir un flux de valeurs `receive_as_json` entrantes . `FromServer` Ensuite, il utilise une `while let` boucle pour gérer les réponses entrantes, en vérifiant les résultats d'erreur et en imprimant chaque réponse du serveur pour que l'utilisateur puisse la voir..

La fonction principale du client

Puisque nous avons présenté à la fois `send_commands` et `handle_replies` , nous pouvons montrer la fonction principale du client de chat, depuis *src/bin/client.rs* :

```
use async_std::task;

fn main() -> ChatResult<()> {
    let address = std:: env::args().nth(1)
        .expect("Usage: client ADDRESS:PORT");

    task:: block_on(async {
        let socket = net:: TcpStream::connect(address).await?;
        socket.set_nodelay(true)?;

        let to_server = send_commands(socket.clone());
        let from_server = handle_replies(socket);

        from_server.race(to_server).await?;
```

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

Après avoir obtenu l'adresse du serveur à partir de la ligne de commande, `main` a une série de fonctions asynchrones qu'il aimerait appeler, il encapsule donc le reste de la fonction dans un bloc asynchrone et passe le futur du bloc `async_std::task::block_on` à s'exécuter.

Une fois la connexion établie, nous voulons que les fonctions `send_commands` et `handle_replies` s'exécutent en tandem, afin que nous puissions voir les messages des autres arriver pendant que nous tapons. Si nous entrons dans l'indicateur de fin de fichier ou si la connexion au serveur tombe, le programme doit se fermer.

Compte tenu de ce que nous avons fait ailleurs dans le chapitre, vous pourriez vous attendre à un code comme celui-ci :

```
let to_server = task:: spawn(send_commands(socket.clone()));
let from_server = task::spawn(handle_replies(socket));

to_server.await?;
from_server.await?;
```

Mais puisque nous attendons les deux poignées de jointure, cela nous donne un programme qui se termine une fois *les deux* tâches terminées. Nous voulons sortir dès que l'un *ou* l'autre a terminé. La `race` méthode sur les contrats à terme accomplit cela. L'appel `from_server.race(to_server)` renvoie un nouveau futur qui interroge à la fois `from_server` et `to_server` et revient `Poll::Ready(v)` dès que l'un d'eux est prêt. Les deux contrats à terme doivent avoir le même type de sortie : la valeur finale est celle du futur qui s'est terminé en premier. Le futur inachevé est abandonné.

La `race` méthode, ainsi que de nombreux autres utilitaires pratiques, est défini sur le `async_std::prelude::FutureExt` trait, ce qui `async_std::prelude` nous rend visible.

À ce stade, la seule partie du code du client que nous n'avons pas montrée est la `parse_command` fonction. C'est un code de traitement de texte assez simple, nous ne montrerons donc pas sa définition ici. Voir le code complet dans le référentiel Git pour plus de détails.

La fonction principale du serveur

Voici l'intégralité du contenudu fichier principal du serveur, *src/bin/server/main.rs* :

```
use async_std:: prelude:: *;
use async_chat:: utils:: ChatResult;
use std:: sync::Arc;

mod connection;
mod group;
mod group_table;

use connection::serve;

fn main() -> ChatResult<()> {
    let address = std:: env::args().nth(1).expect("Usage: server ADDRESS");

    let chat_group_table = Arc:: new(group_table:: GroupTable::new());

    async_std:: task:: block_on(async {
        // This code was shown in the chapter introduction.
        use async_std::{net, task};

        let listener = net:: TcpListener::bind(address).await?;

        let mut new_connections = listener.incoming();
        while let Some(socket_result) = new_connections.next().await {
            let socket = socket_result?;
            let groups = chat_group_table.clone();
            task::spawn(async {
                log_error(serve(socket, groups).await);
            });
        }

        Ok(())
    })
}

fn log_error(result:ChatResult<()>) {
    if let Err(error) = result {
        eprintln!("Error: {}", error);
    }
}
```

La `main` fonction du serveur ressemble à celle du client : il effectue un peu de configuration, puis appelle `block_on` pour exécuter un bloc asynchrone qui fait le vrai travail. Pour gérer les connexions entrantes des

clients, il crée un `TcpListener` socket, dont la `incoming` méthode renvoie un flux de `std::io::Result<TcpStream>` valeurs.

Pour chaque connexion entrante, nous générons une tâche asynchrone exécutant la `connection::serve` fonction. Chaque tâche reçoit également une référence à une `GroupTable` valeur représentant la liste actuelle des groupes de discussion de notre serveur, partagée par toutes les connexions via un `Arc` pointeur à références comptées.

Si `connection::serve` renvoie une erreur, nous enregistrons un message dans la sortie d'erreur standard et laissons la tâche se terminer. Les autres connexions continuent de fonctionner normalement.

Gestion des connexions de chat : mutex asynchrones

Voici le cheval de bataille du serveur: la `serve` fonction du `connection` module dans *src/bin/server/connection.rs* :

```
use async_chat:: {FromClient, FromServer};
use async_chat:: utils:: {self, ChatResult};
use async_std:: prelude:: *;
use async_std:: io:: BufReader;
use async_std:: net:: TcpStream;
use async_std:: sync::Arc;

use crate:: group_table::GroupTable;

pub async fn serve(socket: TcpStream, groups: Arc<GroupTable>)
                   -> ChatResult<()>
{
    let outbound = Arc:: new(Outbound::new(socket.clone()));

    let buffered = BufReader:: new(socket);
    let mut from_client = utils::receive_as_json(buffered);
    while let Some(request_result) = from_client.next().await {
        let request = request_result?;

        let result = match request {
            FromClient::Join { group_name } => {
                let group = groups.get_or_create(group_name);
                group.join(outbound.clone());
                Ok(())
            }

            FromClient::Post { group_name, message } => {
                match groups.get(&group_name) {
                    Some(group) => {
```

```
                            group.post(message);
                            Ok(())
                        }
                        None => {
                            Err(format!("Group '{}' does not exist", group_name
                        }
                    }
                }
            };

            if let Err(message) = result {
                let report = FromServer::Error(message);
                outbound.send(report).await?;
            }
        }

        Ok(())
    }
```

C'est presque une image miroir de la `handle_replies` fonction du client : la majeure partie du code est une boucle gérant un flux entrant de `FromClient` valeurs, construit à partir d'un flux TCP mis en mémoire tampon avec `receive_as_json` . Si une erreur se produit, nous générons un `FromServer::Error` paquet pour transmettre la mauvaise nouvelle au client.

En plus des messages d'erreur, les clients aimeraient également recevoir des messages des groupes de discussion qu'ils ont rejoints, de sorte que la connexion au client doit être partagée avec chaque groupe. Nous pourrions simplement donner à chacun un clone du `TcpStream` , mais si deux de ces sources essayaient d'écrire un paquet sur le socket en même temps, leur sortie pourrait être entrelacée et le client finirait par recevoir du JSON brouillé. Nous devons organiser un accès simultané sécurisé à la connexion.

Ceci est géré avec le `Outbound` type, défini dans *src/bin/server/connection.rs* comme suit :

```
use async_std:: sync::Mutex;

pub struct Outbound(Mutex<TcpStream>);

impl Outbound {
    pub fn new(to_client: TcpStream) -> Outbound {
        Outbound(Mutex::new(to_client))
    }
```

```
        pub async fn send(&self, packet: FromServer) -> ChatResult<()> {
            let mut guard = self.0.lock().await;
            utils::send_as_json(&mut *guard, &packet).await?;
            guard.flush().await?;
            Ok(())
        }
    }
```

Une fois créée, une `Outbound` valeur prend possession de a `TcpStream` et l'enveloppe dans a `Mutex` pour s'assurer qu'une seule tâche peut l'utiliser à la fois. La `serve` fonction enveloppe chacun d'eux `Outbound` dans un `Arc` pointeur à références comptées afin que tous les groupes auxquels le client se joint puissent pointer vers la même `Outbound` instance partagée.

Un appel à `Outbound::send` first verrouille le mutex, renvoyant une valeur de garde qui déréférence à l' `TcpStream` intérieur. Nous utilisons `send_as_json` pour transmettre `packet` , puis nous appelons finalement pour nous `guard.flush()` assurer qu'il ne languira pas à moitié transmis dans un tampon quelque part. (À notre connaissance, `TcpStream` ne met pas réellement les données en mémoire tampon, mais le `Write` trait permet à ses implémentations de le faire, nous ne devrions donc pas prendre de risques.)

L'expression `&mut *guard` nous permet de contourner le fait que Rust n'applique pas de contraintes de deref pour respecter les limites des traits. Au lieu de cela, nous déréférencons explicitement la garde du mutex, puis empruntons une référence mutable à celle `TcpStream` qu'elle protège, produisant la `&mut TcpStream` qui l' `send_as_json` exige.

Notez qu'il `Outbound` utilise le `async_std::sync::Mutex` type, pas celui de la bibliothèque standard `Mutex` . Il y a trois raisons à cela.

Tout d'abord, la bibliothèque standard `Mutex` peut mal se comporter si une tâche est suspendue alors qu'elle maintient une protection mutex. Si le thread qui exécutait cette tâche sélectionne une autre tâche qui essaie de verrouiller le même `Mutex` , des problèmes surviennent : du `Mutex` point de vue de , le thread qui le possède déjà essaie de le verrouiller à nouveau. La norme `Mutex` n'est pas conçue pour gérer ce cas, donc elle panique ou se bloque. (Il n'accordera jamais le verrou de manière inappropriée.) Des travaux sont en cours pour que Rust détecte ce problème au moment de la compilation et émette un avertissement chaque fois qu'un `std::sync::Mutex` garde est actif sur une `await` expression. Puisqu'il `Outbound::send` doit maintenir le verrou en atten-

dant le futur de `send_as_json` et `guard.flush`, il doit utiliser celui `async_std` de `Mutex`.

Deuxièmement, la méthode asynchrone `Mutex` renvoie `lock` un futur d'un garde, donc une tâche attendant de verrouiller un mutex cède son thread pour que d'autres tâches l'utilisent jusqu'à ce que le mutex soit prêt. (Si le mutex est déjà disponible, le `lock` futur est prêt immédiatement, et la tâche ne se suspend pas du tout.) La méthode standard `Mutex`, `lock` d'autre part, épingle l'intégralité du thread en attendant d'acquérir le verrou. Étant donné que le code précédent contient le mutex pendant qu'il transmet un paquet sur le réseau, cela peut prendre un certain temps.

Enfin, la norme `Mutex` ne doit être déverrouillée que par le même thread qui l'a verrouillée. Pour faire respecter cela, le type de garde du mutex standard n'est pas implémenté `Send` : il ne peut pas être transmis à d'autres threads. Cela signifie qu'un futur contenant une telle garde n'implémente pas lui-même `Send`, et ne peut pas être passé à `spawn` pour s'exécuter sur un pool de threads ; il ne peut être exécuté qu'avec `block_on` ou `spawn_local`. Le garde d'un `async_std` `Mutex` implémente `Send` donc il n'y a aucun problème à l'utiliser dans les tâches engendrées.

La table de groupe : mutex synchrones

Mais la morale de l'histoire n'est pas aussi simple que "Toujours utiliser `async_std::sync::Mutex` en code asynchrone". Souvent, il n'est pas nécessaire d'attendre quoi que ce soit en maintenant un mutex, et le verrou n'est pas maintenu longtemps. Dans de tels cas, la bibliothèque standard `Mutex` peutêtre beaucoup plus efficace. Le type de notre serveur de chat `GroupTable` illustre ce cas. Voici le contenu complet de *src/bin/server/group_table.rs* :

```
use crate:: group:: Group;
use std:: collections:: HashMap;
use std:: sync::{Arc, Mutex};

pub struct GroupTable(Mutex<HashMap<Arc<String>, Arc<Group>>>);

impl GroupTable {
    pub fn new() -> GroupTable {
        GroupTable(Mutex:: new(HashMap::new()))
    }

    pub fn get(&self, name: &String) ->Option<Arc<Group>> {
```

```
            self.0.lock()
                .unwrap()
                .get(name)
                .cloned()
        }

        pub fn get_or_create(&self, name: Arc<String>) -> Arc<Group> {
            self.0.lock()
                .unwrap()
                .entry(name.clone())
                .or_insert_with(|| Arc:: new(Group::new(name)))
                .clone()
        }
    }
```

A `GroupTable` est simplement une table de hachage protégée par mutex, mappant les noms des groupes de discussion aux groupes réels, tous deux gérés à l'aide de pointeurs comptés par référence. Les méthodes `get` et `get_or_create` verrouillent le mutex, effectuent quelques opérations de table de hachage, peut-être quelques allocations, et retournent.

Dans `GroupTable`, nous utilisons un vieux simple `std::sync::Mutex`. Il n'y a pas du tout de code asynchrone dans ce module, il n'y a donc pas `await` de s à éviter. En effet, si nous voulions utiliser `async_std::sync::Mutex` ici, nous aurions besoin de faire `get` et `get_or_create` dans des fonctions asynchrones, ce qui introduit le surcoût de futures créations, suspensions et reprises pour peu d'avantages : le mutex n'est verrouillé que pour certaines opérations de hachage et peut-être quelques allocations.

Si notre serveur de chat se retrouvait avec des millions d'utilisateurs et que le `GroupTable` mutex devenait un goulot d'étranglement, le rendre asynchrone ne résoudrait pas ce problème. Il serait probablement préférable d'utiliser une sorte de type de collection spécialisé pour l'accès simultané au lieu de `HashMap`. Par exemple, la `dashmap` caisse fournit un tel type.

Groupes de discussion : chaînes de diffusion de tokio

Dans notre serveur, le `group::Group` genrereprésente un groupe de discussion. Ce type doit uniquement prendre en charge les deux méthodes qui `connection::serve` appellent : `join`, pour ajouter un nouveau membre, et `post`, pour publier un message. Chaque message posté doit être distribué à tous les membres.

C'est là que nous abordons le défi mentionné précédemment de la contre-*pression* . Il y a plusieurs besoins en tension les uns avec les autres :

- Si un membre ne peut pas suivre les messages envoyés au groupe (s'il a une connexion réseau lente, par exemple), les autres membres du groupe ne devraient pas être affectés.
- Même si un membre prend du retard, il devrait y avoir un moyen pour lui de rejoindre la conversation et de continuer à participer d'une manière ou d'une autre.
- La mémoire utilisée pour la mise en mémoire tampon des messages ne doit pas croître sans limite.

Étant donné que ces défis sont courants lors de la mise en œuvre de modèles de communication plusieurs à plusieurs, la `tokio` caisse fournit un type de *canal de diffusion* qui met en œuvre un ensemble raisonnable de compromis. Un `tokio` canal de diffusion est une file d'attente de valeurs (dans notre cas, des messages de chat) qui permet à un nombre quelconque de threads ou de tâches différents d'envoyer et de recevoir des valeurs. C'est ce qu'on appelle un canal de « diffusion », car chaque consommateur reçoit sa propre copie de chaque valeur envoyée. (Le type de valeur doit implémenter `Clone` .)

Normalement, une chaîne de diffusionconserve un message dans la file d'attente jusqu'à ce que chaque consommateur ait reçu sa copie. Mais si la longueur de la file d'attente dépasse la capacité maximale du canal, spécifiée lors de sa création, les messages les plus anciens sont supprimés. Tous les consommateurs qui ne pouvaient pas suivre reçoivent une erreur la prochaine fois qu'ils tentent de recevoir leur prochain message, et le canal les rattrape jusqu'au message le plus ancien encore disponible.

Par exemple, la Figure 20-4 montre un canal de diffusion avec une capacité maximale de 16 valeurs.

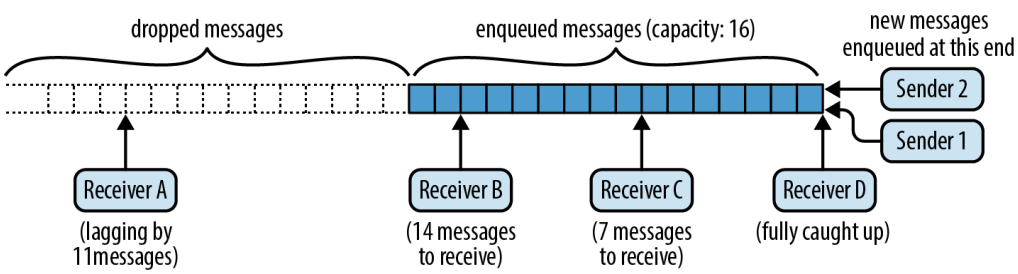

Illustration 20-4. Une chaîne de diffusion tokio

Deux expéditeurs mettent les messages en file d'attente et quatre destinataires les retirent de la file d'attente, ou plus précisément, copient les messages hors de la file d'attente. Le récepteur B a encore 14 messages à recevoir, le récepteur C en a 7 et le récepteur D est entièrement rattrapé. Le

récepteur A a pris du retard et 11 messages ont été abandonnés avant qu'il ne puisse les voir. Sa prochaine tentative de réception d'un message échouera, renvoyant une erreur indiquant la situation, et il sera rattrapé à la fin actuelle de la file d'attente.

Notre serveur de chat représente chaque groupe de chat comme un canal de diffusion porteur de `Arc<String>` valeurs : poster un message au groupe le diffuse à tous les membres actuels. Voici la définition du `group::Group` type, défini dans *src/bin/server/group.rs* :

```
use async_std:: task;
use crate:: connection:: Outbound;
use std:: sync:: Arc;
use tokio:: sync::broadcast;

pub struct Group {
    name: Arc<String>,
    sender: broadcast::Sender<Arc<String>>
}

impl Group {
    pub fn new(name: Arc<String>) -> Group {
        let (sender, _receiver) = broadcast::channel(1000);
        Group { name, sender }
    }

    pub fn join(&self, outbound:Arc<Outbound>) {
        let receiver = self.sender.subscribe();

        task::spawn(handle_subscriber(self.name.clone(),
                                      receiver,
                                      outbound));
    }

    pub fn post(&self, message:Arc<String>) {
        // This only returns an error when there are no subscribers. A
        // connection's outgoing side can exit, dropping its subscription,
        // slightly before its incoming side, which may end up trying to se
        // message to an empty group.
        let _ignored = self.sender.send(message);
    }
}
```

Une `Group` structure contient le nom du groupe de discussion, ainsi qu'un `broadcast::Sender` représentant l'extrémité d'envoi du canal de diffusion du groupe . La `Group::new` fonction appelle `broadcast::channel` à créer un canal de diffusion avec une capacité maximale de 1 000 messages. La `channel` fonction renvoie à la fois un

expéditeur et un destinataire, mais nous n'avons pas besoin du destinataire à ce stade, puisque le groupe n'a pas encore de membres.

Pour ajouter un nouveau membre au groupe, la `Group::join` méthode appelle la méthode de l'expéditeur `subscribe` pour créer un nouveau récepteur pour le canal. Ensuite, il génère une nouvelle tâche asynchrone pour surveiller ce récepteur pour les messages et les réécrire au client, dans la `handle_subscribe` fonction.

Avec ces détails en main, la `Group::post` méthode est simple : elle envoie simplement le message au canal de diffusion. Étant donné que les valeurs transportées par le canal sont des `Arc<String>` valeurs, donner à chaque récepteur sa propre copie d'un message augmente simplement le nombre de références du message, sans aucune copie ni allocation de tas. Une fois que tous les abonnés ont transmis le message, le compteur de références tombe à zéro et le message est libéré.

Voici la définition de `handle_subscriber` :

```
use async_chat:: FromServer;
use tokio:: sync:: broadcast:: error::RecvError;

async fn handle_subscriber(group_name: Arc<String>,
                           mut receiver: broadcast:: Receiver<Arc<String>>,
                           outbound: Arc<Outbound>)
{
    loop {
        let packet = match receiver.recv().await {
            Ok(message) => FromServer:: Message {
                group_name: group_name.clone(),
                message:message.clone(),
            },

            Err(RecvError:: Lagged(n)) => FromServer::Error(
                format!("Dropped {} messages from {}.", n, group_name)
            ),

            Err(RecvError::Closed) => break,
        };

        if outbound.send(packet).await.is_err() {
            break;
        }
    }
}
```

Bien que les détails soient différents, la forme de cette fonction est familière : c'est une boucle qui reçoit les messages du canal de diffusion et les retransmet au client via la `Outbound` valeur partagée. Si la boucle ne peut pas suivre le canal de diffusion, elle reçoit une `Lagged` erreur, qu'elle signale consciencieusement au client.

Si le renvoi d'un paquet au client échoue complètement, peut-être parce que la connexion s'est fermée, `handle_subscriber` quitte sa boucle et revient, provoquant la fermeture de la tâche asynchrone. Cela supprime le canal de diffusion `Receiver` , en le désabonnant du canal. De cette façon, lorsqu'une connexion est abandonnée, chacune de ses appartenances au groupe est nettoyée la prochaine fois que le groupe essaie de lui envoyer un message.

Nos groupes de discussion ne se ferment jamais, car nous ne supprimons jamais un groupe de la table des groupes, mais juste pour être complet, nous sommes `handle_subscriber` prêts à gérer une `Closed` erreur en quittant la tâche.

Notez que nous créons une nouvelle tâche asynchrone pour chaque appartenance à un groupe de chaque client. Cela est possible car les tâches asynchrones utilisent beaucoup moins de mémoire que les threads et parce que le passage d'une tâche asynchrone à une autre au sein d'un processus est assez efficace.

Ceci est donc le code complet du serveur de chat. C'est un peu spartiate, et il y a beaucoup plus de fonctionnalités intéressantes dans les `async_std` , `tokio` et `futures` crates que nous ne pouvons couvrir dans ce livre, mais idéalement cet exemple étendu parvient à illustrer comment certaines des fonctionnalités de l'écosystème asynchrone fonctionnent ensemble : tâches asynchrones, flux , les traits d'E/S asynchrones, les canaux et les mutex des deux versions.

Futurs primitifs et exécuteurs : quand un futur mérite-t-il à nouveau d'être interrogé ?

La discussionserver montre comment nous pouvons écrire du code en utilisant des primitives asynchrones comme `TcpListener` et le `broadcast` canal, et utiliser des exécuteurs comme `block_on` et `spawn` pour piloter leur exécution. Maintenant, nous pouvons jeter un œil à la façon dont ces choses sont mises en œuvre. La question clé est,

quand un futur revient `Poll::Pending`, comment se coordonne-t-il avec l'exécuteur testamentaire pour l'interroger à nouveau au bon moment ?

Pensez à ce qui se passe lorsque nous exécutons un code comme celui-ci, à partir de la fonction du client de chat `main` :

```
task:: block_on(async {
    let socket = net:: TcpStream::connect(address).await?;
    ...
})
```

Les premiers `block_on` sondagesl'avenir du bloc asynchrone, la connexion réseau n'est presque certainement pas prête immédiatement, donc `block_on` se met en veille. Mais quand devrait-il se réveilleren haut? D'une manière ou d'une autre, une fois que la connexion réseau est prête, il `TcpStream` doit indiquer `block_on` qu'il doit réessayer d'interroger l'avenir du bloc asynchrone, car il sait que cette fois, le `await` sera terminé et que l'exécution du bloc asynchrone peut progresser.

Lorsqu'un exécuteur comme `block_on` interroge un futur, il doit passer un rappel appelé un *waker* . Si l'avenir n'est pas encore prêt, les règles du `Future` trait disent qu'il doit revenir `Poll::Pending` pour le moment et faire en sorte que l'éveilleur soit invoqué plus tard, si et quand l'avenir vaut la peine d'être interrogé à nouveau.

Ainsi, une implémentation manuscrite de `Future` ressemble souvent à ceci :

```
use std:: task::Waker;

struct MyPrimitiveFuture {
    ...
    waker:Option<Waker>,
}

impl Future for MyPrimitiveFuture {
    type Output = ...;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) ->Poll<...> {
        ...

        if ... future is ready ... {
            return Poll::Ready(final_value);
        }
```

```
            // Save the waker for later.
            self.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
```

En d'autres termes, si la valeur du futur est prête, retournez-la. Sinon, `Context` placez quelque part un clone de l'éveilleur de et retournez `Poll::Pending`.

Lorsque le futur vaut la peine d'être interrogé à nouveau, le futur doit notifier le dernier exécuteur qui l'a interrogé en appelant la `wake` méthode de son éveilleur :

```
// If we have a waker, invoke it, and clear `self.waker`.
if let Some(waker) = self.waker.take() {
    waker.wake();
}
```

Idéalement, l'exécuteur et le futur interrogent et se réveillent à tour de rôle: l'exécuteur interroge le futur et s'endort, puis le futur invoque le réveil, de sorte que l'exécuteur se réveille et interroge à nouveau le futur.

Les futurs des fonctions et des blocs asynchrones ne traitent pas des wakers eux-mêmes. Ils transmettent simplement le contexte qui leur est donné aux sous-futurs qu'ils attendent, leur déléguant l'obligation de sauver et d'invoquer des éveilleurs. Dans notre client de chat, le premier sondage sur l'avenir du bloc asynchrone transmet simplement le contexte lorsqu'il attend `TcpStream::connect` l'avenir de . Les sondages suivants transmettent de la même manière leur contexte à l'avenir que le bloc attend ensuite.

`TcpStream::connect` Le futur de gère l'interrogation comme indiqué dans l'exemple précédent : il transmet l'éveilleur à un thread d'assistance qui attend que la connexion soit prête, puis appelle l'éveilleur.

`Waker` implémente `Clone` et `Send`, ainsi un futur peut toujours faire sa propre copie du waker et l'envoyer à d'autres threads si nécessaire. La `Waker::wake` méthode consomme le waker. Il existe également une `wake_by_ref` méthode qui ne le fait pas, mais certains exécuteurs peuvent implémenter la version consommatrice un peu plus efficacement. (La différence est au plus de `clone`.)

Il est inoffensif pour un exécuteur de surinterroger un futur, simplement inefficace. Les contrats à terme, cependant, devraient veiller à n'invoquer

un réveil que lorsque l'interrogation ferait des progrès réels : un cycle de réveils et d'interrogations parasites peut empêcher un exécuteur de dormir, gaspillant de l'énergie et laissant le processeur moins réactif aux autres tâches.

Maintenant que nous avons montré comment les exécuteurs et les futurs primitifs communiquent, nous allons implémenter nous-mêmes un futur primitif, puis parcourir une implémentation de l' `block_on` exécuteur.

Invocation de Wakers : spawn_blocking

Plus tôtdans le chapitre, nous avons décrit la `spawn_blocking` fonction, qui démarre une fermeture donnée s'exécutant sur un autre thread et renvoie un futur de sa valeur de retour. Nous avons maintenant toutes les pièces dont nous avons besoin pour `spawn_blocking` nous mettre en œuvre. Pour plus de simplicité, notre version crée un nouveau thread pour chaque fermeture, plutôt que d'utiliser un pool de threads, comme `async_std` le fait la version de .

Bien que `spawn_blocking` renvoie un futur, nous n'allons pas l'écrire comme un `async fn` . Il s'agira plutôt d'une fonction ordinaire et synchrone qui renvoie une structure, `SpawnBlocking` , sur laquelle nous nous implémenterons `Future` nous-mêmes.

La signature de notre `spawn_blocking` est la suivante :

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
      F: Send + 'static,
      T:Send + 'static,
```

Puisque nous devons envoyer la fermeture à un autre thread et ramener la valeur de retour, la fermeture `F` et sa valeur de retour `T` doivent implémenter `Send` . Et comme nous n'avons aucune idée de la durée de fonctionnement du thread, ils doivent l'être tous les deux `'static` également. Ce sont les mêmes limites que lui- `std::thread::spawn` même impose.

`SpawnBlocking<T>` est un futur de la valeur de retour de la fermeture. Voici sa définition :

```
use std:: sync:: {Arc, Mutex};
use std:: task::Waker;

pub struct SpawnBlocking<T>(Arc<Mutex<Shared<T>>>);
```

```
struct Shared<T> {
    value: Option<T>,
    waker:Option<Waker>,
}
```

La `Shared` structuredoit servir de rendez-vous entre le futur et le thread exécutant la fermeture, il appartient donc à un `Arc` et est protégé par un `Mutex` . (Un mutex synchrone convient ici.) L'interrogation du futur vérifie si `value` est présent et enregistre le réveil dans le `waker` cas contraire. Le thread qui exécute la fermeture enregistre sa valeur de retour dans `value` puis appelle `waker` , s'il est présent.

Voici la définition complète de `spawn_blocking` :

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
      F: Send + 'static,
      T: Send + 'static,
{
    let inner = Arc:: new(Mutex:: new(Shared {
        value: None,
        waker:None,
    }));

    std:: thread::spawn({
        let inner = inner.clone();
        move || {
            let value = closure();

            let maybe_waker = {
                let mut guard = inner.lock().unwrap();
                guard.value = Some(value);
                guard.waker.take()
            };

            if let Some(waker) = maybe_waker {
                waker.wake();
            }
        }
    });

    SpawnBlocking(inner)
}
```

Après avoir créé la `Shared` valeur, cela génère un thread pour exécuter la fermeture, stocker le résultat dans le champ `Shared` 's `value` et invoquer le waker, le cas échéant.

Nous pouvons implémenter `Future` pour `SpawnBlocking` comme suit:

```
use std:: future:: Future;
use std:: pin:: Pin;
use std:: task::{Context, Poll};

impl<T:Send> Future for SpawnBlocking<T> {
    type Output = T;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<T> {
        let mut guard = self.0.lock().unwrap();
        if let Some(value) = guard.value.take() {
            return Poll::Ready(value);
        }

        guard.waker = Some(cx.waker().clone());
        Poll::Pending
    }
}
```

L' interrogation d'un `SpawnBlocking` vérifie si la valeur de la fermeture est prête, prend possession et la renvoie si c'est le cas. Sinon, le futur est toujours en attente, il enregistre donc un clone de l'éveil du contexte dans le `waker` champ du futur.

Une fois que `Future` a est revenu `Poll::Ready` , vous n'êtes pas censé l'interroger à nouveau. Les façons habituelles de consommer des contrats à terme, comme `await` et `block_on` , respectent toutes cette règle. Si un `SpawnBlocking` avenir est dépassé, rien de particulièrement terrible ne se produit, mais il ne fait aucun effort pour gérer ce cas non plus. Ceci est typique des contrats à terme manuscrits.

Implémentation de block_on

en outrepour pouvoir implémenter des futurs primitifs, nous avons également toutes les pièces dont nous avons besoin pour construire un exécuteur simple. Dans cette section, nous allons écrire notre propre version de `block_on` . Ce sera un peu plus simple que `async_std` la version de ; par exemple, il ne prend pas en charge `spawn_local` , les variables locales de tâche ou les invocations imbriquées (appels `block_on` à partir de code asynchrone). Mais c'est suffisant pour exécuter notre client et serveur de chat.

Voici le code :

```rust
use waker_fn:: waker_fn;      // Cargo.toml: waker-fn = "1.1"
use futures_lite:: pin;       // Cargo.toml: futures-lite = "1.11"
use crossbeam:: sync:: Parker; // Cargo.toml: crossbeam = "0.8"
use std:: future:: Future;
use std:: task::{Context, Poll};

fn block_on<F: Future>(future: F) -> F:: Output {
    let parker = Parker:: new();
    let unparker = parker.unparker().clone();
    let waker = waker_fn(move || unparker.unpark());
    let mut context = Context::from_waker(&waker);

    pin!(future);

    loop {
        match future.as_mut().poll(&mut context) {
            Poll:: Ready(value) => return value,
            Poll::Pending => parker.park(),
        }
    }
}
```

C'est assez court, mais il se passe beaucoup de choses, alors prenons-le un morceau à la fois.

```rust
let parker = Parker::new();
let unparker = parker.unparker().clone();
```

Le type `crossbeam` de crate `Parker` est une simple primitive bloquante : l'appel `parker.park()` bloque le thread jusqu'à ce que quelqu'un d'autre appelle `.unpark()` le correspondant `Unparker`, que vous obtenez au préalable en appelant `parker.unparker()`. Si vous avez `unpark` un thread qui n'est pas encore parqué, son prochain appel à `park` revient immédiatement, sans blocage. Notre `block_on` utilisera le `Parker` pour attendre chaque fois que le futur n'est pas prêt, et le réveil que nous passons au futur le débloquera.

```rust
let waker = waker_fn(move || unparker.unpark());
```

La `waker_fn` fonction, à partir de la caisse du même nom, crée un `Waker` à partir d'une fermeture donnée. Ici, nous faisons un `Waker` qui, lorsqu'il est invoqué, appelle la fermeture `move || unparker.unpark()`. Vous pouvez également créer des wakers en implémentant le `std::task::Wake` trait, mais `waker_fn` c'est un peu plus pratique ici.

```
pin!(future);
```

Étant donné une variable contenant un futur de type `F`, la `pin!` macro s'approprie le futur et déclare une nouvelle variable de même nom dont le type est `Pin<&mut F>` et qui emprunte le futur. Cela nous donne le `Pin<&mut Self>` requis par la `poll` méthode. Pour des raisons que nous expliquerons dans la section suivante, les futurs des fonctions et des blocs asynchrones doivent être référencés via a `Pin` avant de pouvoir être interrogés.

```
loop {
    match future.as_mut().poll(&mut context) {
        Poll:: Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

Enfin, la boucle de sondage est assez simple. Passant un contexte portant notre réveil, nous interrogeons le futur jusqu'à ce qu'il revienne `Poll::Ready`. S'il retourne `Poll::Pending`, nous garons le thread, qui se bloque jusqu'à ce qu'il `waker` soit appelé. Puis nous réessayons.

L' `as_mut` appel nous permet d'interroger `future` sans renoncer à la propriété ; nous expliquerons cela plus en détail dans la section suivante.

Épingler

Bien queles fonctions et les blocs asynchrones sont essentiels pour écrire du code asynchrone clair, la gestion de leur avenir nécessite un peu de prudence. Le `Pin` type aide Rust à s'assurer qu'ils sont utilisés en toute sécurité.

Dans cette section, nous montrerons pourquoi les contrats à terme d'appels de fonction et de blocs asynchrones ne peuvent pas être gérés aussi librement que les valeurs Rust ordinaires. Ensuite, nous montrerons comment `Pin` sert de «sceau d'approbation» sur des pointeurs sur lesquels on peut compter pour gérer ces contrats à terme en toute sécurité. Enfin, nous montrerons quelques façons de travailler avec des `Pin` valeurs.

Les deux étapes de la vie d'un avenir

Considérez cette fonction asynchrone simple :

```
use async_std:: io:: prelude:: *;
use async_std::{io, net};

async fn fetch_string(address: &str) -> io:: Result<String> {
    ❶
    let mut socket = net:: TcpStream:: connect(address).await❷?;
    let mut buf = String::new();
    socket.read_to_string(&mut buf).await❸?;
    Ok(buf)
}
```

Cela ouvre une connexion TCP à l'adresse donnée et renvoie, sous forme de `String`, tout ce que le serveur veut envoyer. Les points étiquetés ❶, ❷ et ❸ sont les points de *reprise*, les points dans le code de la fonction asynchrone auxquels l'exécution peut être suspendue.

Supposons que vous l'appeliez, sans attendre, comme ceci :

```
let response = fetch_string("localhost:6502");
```

Now `response` est un futur prêt à commencer l'exécution au début de `fetch_string`, avec l'argument donné. En mémoire, le futur ressemble à la figure 20-5.

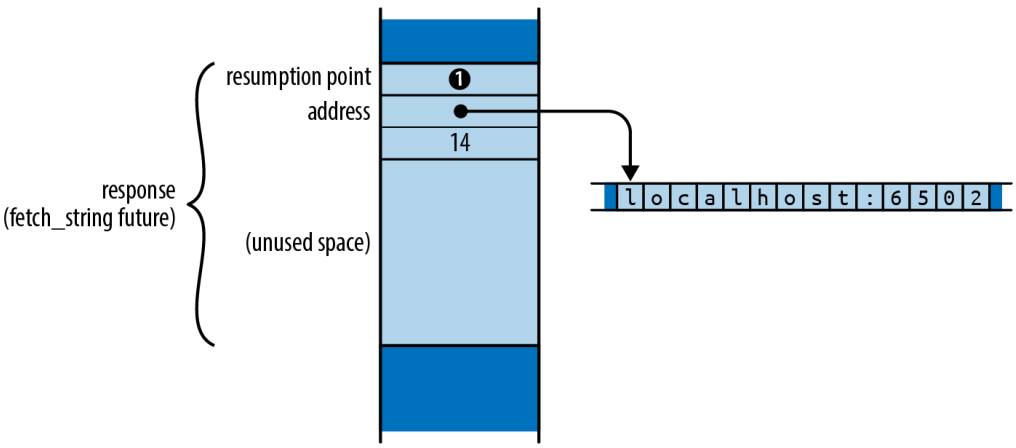

Illustration 20-5. L'avenir construit pour un appel à `fetch_string`

Puisque nous venons de créer ce futur, il est indiqué que l'exécution doit commencer au point de reprise ❶, en haut du corps de la fonction. Dans cet état, les seules valeurs dont un futur a besoin pour continuer sont les arguments de la fonction.

Supposons maintenant que vous interrogez `response` plusieurs fois et qu'il atteigne ce point dans le corps de la fonction :

```
socket.read_to_string(&mut buf).await❸?;
```

Supposons en outre que le résultat de `read_to_string` n'est pas prêt, donc le sondage renvoie `Poll::Pending` . À ce stade, le futur ressemble à la Figure 20-6 .

Un futur doit toujours contenir toutes les informations nécessaires pour reprendre l'exécution la prochaine fois qu'il est interrogé. Dans ce cas c'est :

- Point de reprise ❸, disant que l'exécution devrait reprendre dans le futur du `await` scrutin . `read_to_string`
- Les variables actives à ce point de reprise : `socket` et `buf` . La valeur de `address` n'est plus présente dans le futur, puisque la fonction n'en a plus besoin.
- Le `read_to_string` sous-futur, dont l' `await` expression est en pleine vogue.

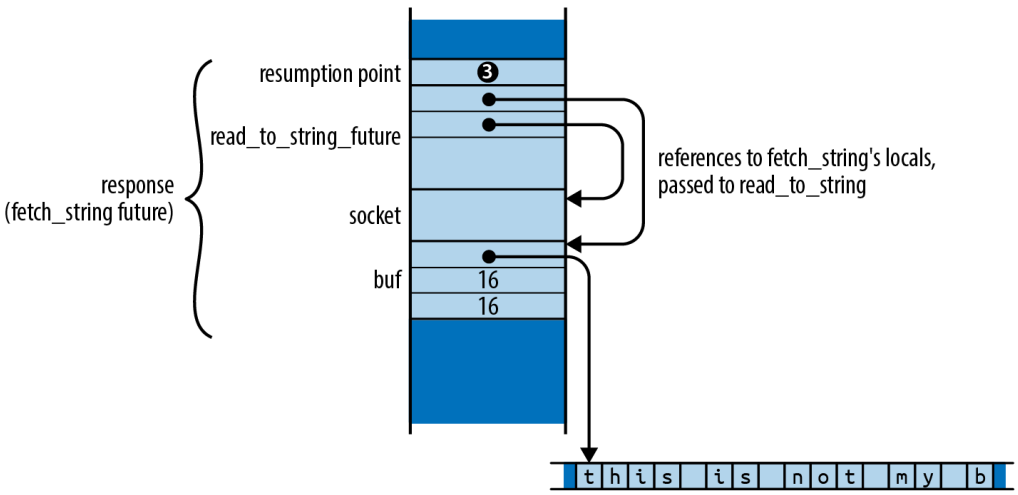

Illustration 20-6. Le même avenir, au milieu de l'attente `read_to_string`

Notez que l'appel à `read_to_string` a empruntéréférences à `socket` et `buf` . Dans une fonction synchrone, toutes les variables locales vivent sur la pile, mais dans une fonction asynchrone, les variables locales qui sont actives sur an `await` doivent être localisées dans le futur, elles seront donc disponibles lorsqu'elle sera à nouveau interrogée. Emprunter une référence à une telle variable emprunte une partie du futur.

Cependant, Rust exige que les valeurs ne soient pas déplacées pendant qu'elles sont empruntées. Supposons que vous deviez déplacer ce futur vers un nouvel emplacement :

```
let new_variable = response;
```

Rust n'a aucun moyen de trouver toutes les références actives et de les ajuster en conséquence. Au lieu de pointer vers `socket` et `buf` vers leurs nouveaux emplacements, les références continuent de pointer vers leurs

anciens emplacements dans le fichier `response`. Ils sont devenus des pointeurs pendants, comme le montre la Figure 20-7.

Empêcher les valeurs empruntées d'être déplacées est généralement la responsabilité du vérificateur d'emprunt. Le vérificateur d'emprunt traite les variables comme les racines des arbres de propriété, mais contrairement aux variables stockées sur la pile, les variables stockées dans les contrats à terme sont déplacées si le futur lui-même se déplace. Cela signifie que les emprunts de `socket` et `buf` affectent non seulement ce qui `fetch_string` peut être fait avec ses propres variables, mais aussi ce que son appelant peut faire en toute sécurité avec `response`, le futur qui les détient. L'avenir des fonctions asynchrones est un angle mort pour le vérificateur d'emprunt, que Rust doit couvrir d'une manière ou d'une autre s'il veut tenir ses promesses de sécurité de la mémoire.

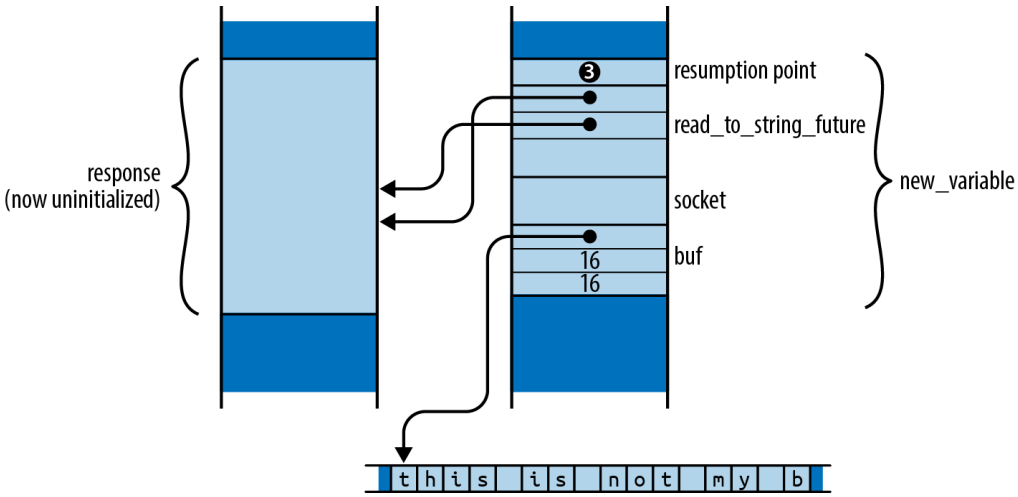

Illustration 20-7. `fetch_string` 's future, déplacé pendant l'emprunt (Rust empêche cela)

La solution de Rust à ce problème repose sur l'idée que les contrats à terme sont toujours sûrs à déplacer lorsqu'ils sont créés pour la première fois et ne deviennent dangereux à déplacer que lorsqu'ils sont interrogés. Un futur qui vient d'être créé en appelant une fonction asynchrone contient simplement un point de reprise et les valeurs des arguments. Celles-ci ne concernent que le corps de la fonction asynchrone, qui n'a pas encore commencé son exécution. Seul l'interrogation d'un futur peut emprunter son contenu.

À partir de là, nous pouvons voir que chaque futur a deux étapes de vie :

- La première étape commence lorsque l'avenir est créé. Étant donné que le corps de la fonction n'a pas commencé son exécution, aucune partie de celle-ci ne peut encore être empruntée. À ce stade, il est aussi sûr de se déplacer que n'importe quelle autre valeur de Rust.
- La deuxième étape commence la première fois que le futur est interrogé. Une fois que le corps de la fonction a commencé son exécution, il

peut emprunter des références à des variables stockées dans le futur, puis attendre, laissant cette partie du futur empruntée. À partir de son premier sondage, nous devons supposer que l'avenir n'est peut-être pas sûr.

La flexibilité de la première étape de la vie est ce qui nous permet de passer les contrats à terme à `block_on` et `spawn` et d'appeler des méthodes d'adaptation comme `race` et `fuse`, qui prennent toutes les contrats à terme par valeur. En fait, même l'appel de fonction asynchrone qui a créé le futur en premier lieu devait le renvoyer à l'appelant ; c'était un mouvement aussi.

Pour entrer dans sa deuxième étape de vie, l'avenir doit être interrogé. La `poll` méthode nécessite que le futur soit passé en tant que `Pin<&mut Self>` valeur. `Pin` est un wrapper pour les types de pointeurs (comme `&mut Self`) qui limite la façon dont les pointeurs peuvent être utilisés, garantissant que leurs référents (comme `Self`) ne peuvent plus jamais être déplacés. Vous devez donc produire un `Pin` pointeur encapsulé vers le futur avant de pouvoir l'interroger.

Voici donc la stratégie de Rust pour assurer la sécurité des futurs : un futur ne peut pas devenir dangereux à déplacer tant qu'il n'est pas interrogé ; vous ne pouvez pas interroger un futur tant que vous n'avez pas construit un `Pin` pointeur encapsulé vers celui-ci ; et une fois que vous avez fait cela, l'avenir ne peut pas être déplacé.

"Une valeur que vous ne pouvez pas déplacer" semble impossible : les déplacements sont partout dans Rust. Nous expliquerons exactement comment `Pin` protège les contrats à terme dans la section suivante.

Bien que cette section ait traité des fonctions asynchrones, tout ici s'applique également aux blocs asynchrones. Un futur fraîchement créé d'un bloc asynchrone capture simplement les variables qu'il utilisera à partir du code environnant, comme une fermeture. Seul l'interrogation du futur peut créer des références à son contenu, le rendant dangereux à déplacer.

Gardez à l'esprit que cette fragilité de mouvement est limitée aux futures des fonctions et des blocs asynchrones, avec leurs `Future` implémentations spéciales générées par le compilateur. Si vous implémentez `Future` à la main pour vos propres types, comme nous l'avons fait pour notre `SpawnBlocking` type dans "Invoking Wakers: spawn_blocking", ces futurs sont parfaitement sûrs à déplacer à la fois avant et après avoir été interrogés. Dans toute implémentation manuscrite `poll`, le vérificateur d'emprunt garantit que toutes les références que vous avez emprun-

tées à des parties de `self` ont disparu au moment du `poll` retour. Ce n'est que parce que les fonctions et les blocs asynchrones ont le pouvoir de suspendre l'exécution au milieu d'un appel de fonction, avec des emprunts en cours, que nous devons gérer leur avenir avec soin.

Pointeurs épinglés

Le `Pin` typeest un emballagepour les pointeurs vers des contrats à terme qui limitent la façon dont les pointeurs peuvent être utilisés pour s'assurer que les contrats à terme ne peuvent pas être déplacés une fois qu'ils ont été interrogés. Ces restrictions peuvent être levées pour les futurs qui ne craignent pas d'être déplacés, mais elles sont essentielles pour interroger en toute sécurité les futurs des fonctions et des blocs asynchrones.

Par *pointeur* , nous entendons tout type qui implémente `Deref` , et éventuellement `DerefMut` . Un `Pin` pointeur enroulé autour d'un pointeur est appelé un *pointeur épinglé* . `Pin<&mut T>` et `Pin<Box<T>>` sont typiques.

La définition de `Pin` dans la bibliothèque standard est simple :

```
pub struct Pin<P> {
    pointer:P,
}
```

Notez que le `pointer` champ n'est *pas* `pub` . Cela signifie que la seule façon de construire ou d'utiliser a `Pin` passe par les méthodes soigneusement choisies fournies par le type.

Étant donné le futur d'une fonction ou d'un bloc asynchrone, il n'y a que quelques façons d'obtenir un pointeur épinglé vers celui-ci :

- La `pin!` macro, depuis le `futures-lite` crate, masque une variable de type `T` avec une nouvelle de type `Pin<&mut T>` . La nouvelle variable pointe vers la valeur d'origine, qui a été déplacée vers un emplacement temporaire anonyme sur la pile. Lorsque la variable sort de la portée, la valeur est supprimée. Nous avons utilisé `pin!` dans notre `block_on` implémentation pour épingler le futur que nous voulions interroger.
- Le `Box::pin` constructeur de la bibliothèque standard s'approprie une valeur de n'importe quel type `T` , la déplace dans le tas et renvoie un `Pin<Box<T>>` .
- `Pin<Box<T>>` implements `From<Box<T>>` , `Pin::from(boxed)` s'approprie donc `boxed` et vous restitue une boîte épinglée pointant vers

la même chose `T` sur le tas.

Chaque façon d'obtenir un pointeur épinglé vers ces contrats à terme implique de renoncer à la propriété du futur, et il n'y a aucun moyen de le récupérer. Le pointeur épinglé lui-même peut être déplacé comme bon vous semble, bien sûr, mais le déplacement d'un pointeur ne déplace pas son référent. Ainsi, la possession d'un pointeur épinglé vers un futur sert de preuve que vous avez définitivement renoncé à la capacité de déplacer ce futur. C'est tout ce dont nous avons besoin pour savoir qu'il peut être interrogé en toute sécurité.

Une fois que vous avez épinglé un futur, si vous souhaitez l'interroger, tous les `Pin<pointer to T>` types ont une `as_mut` méthode qui déréférence le pointeur et renvoie le `Pin<&mut T>` qui l' `poll` exige.

La `as_mut` méthode peut également vous aider à interroger un avenir sans renoncer à la propriété. Notre `block_on` implémentation l'a utilisé dans ce rôle :

```
pin!(future);

loop {
    match future.as_mut().poll(&mut context) {
        Poll:: Ready(value) => return value,
        Poll::Pending => parker.park(),
    }
}
```

Ici, la `pin!` macro a été redéclarée `future` en tant que `Pin<&mut F>` , nous pouvons donc simplement la transmettre à `poll` . Mais les références mutables ne sont pas `Copy` , donc `Pin<&mut F>` ne peuvent pas l'être non `Copy` plus, ce qui signifie que l'appel `future.poll()` direct prendrait possession de `future` , laissant la prochaine itération de la boucle avec une variable non initialisée. Pour éviter cela, nous appelons `future.as_mut()` à réemprunter un nouveau `Pin<&mut F>` pour chaque itération de boucle.

Il n'y a aucun moyen d'obtenir une `&mut` référence à un futur épinglé : si vous le pouviez, vous pourriez utiliser `std::mem::replace` ou `std::mem::swap` pour le déplacer et mettre un futur différent à sa place.

La raison pour laquelle nous n'avons pas à nous soucier d'épingler des contrats à terme dans du code asynchrone ordinaire est que les moyens les plus courants d'obtenir la valeur d'un contrat à terme - l'attendre ou le transmettre à un exécuteur - s'approprient tous le futur et gèrent l'épin-

glage en interne. Par exemple, notre `block_on` implémentation s'approprie l'avenir et utilise la `pin!` macro pour produire le `Pin<&mut F>` nécessaire à interroger. Une `await` expression s'approprie également le futur et utilise une approche similaire à la `pin!` macro en interne.

Le trait de détachement

Cependant, tous les contrats à terme ne nécessitent pas ce type de manipulation prudente. Pour toute implémentation manuscrite d' `Future` un type ordinaire, comme notre `SpawnBlocking` type mentionné précédemment, les restrictions sur la construction et l'utilisation de pointeurs épinglés sont inutiles.

Ces types durables implémentent le `Unpin` marqueurcaractéristique:

```
trait Unpin { }
```

Presque tous les types de Rust implémentent automatiquement `Unpin` , en utilisant un support spécial dans le compilateur. Les fonctions asynchrones et les contrats à terme de blocs sont les exceptions à cette règle.

Pour `Unpin` les types, `Pin` n'impose aucune restriction. Vous pouvez créer un pointeur épinglé à partir d'un pointeur ordinaire avec `Pin::new` et récupérer le pointeur avec `Pin::into_inner` . Le lui-même passe le long des implémentations et des implémentations `Pin` du pointeur . `Deref DerefMut`

Par exemple, `String` implements `Unpin` , nous pouvons donc écrire :

```
let mut string = "Pinned?".to_string();
let mut pinned: Pin<&mut String> = Pin::new(&mut string);

pinned.push_str(" Not");
Pin::into_inner(pinned).push_str(" so much.");

let new_home = string;
assert_eq!(new_home, "Pinned? Not so much.");
```

Même après avoir créé un `Pin<&mut String>` , nous avons un accès mutable complet à la chaîne et pouvons le déplacer vers une nouvelle variable une fois que le `Pin` a été consommé par `into_inner` et que la référence mutable a disparu. Donc, pour les types qui sont `Unpin` - ce qui est presque tous - il `Pin` y a une enveloppe ennuyeuse autour des pointeurs vers ce type.

Cela signifie que lorsque vous implémentez `Future` pour vos propres `Unpin` types, votre `poll` implémentation peut traiter `self` comme s'il s'agissait de `&mut Self` , et non de `Pin<&mut Self>` . L'épinglage devient quelque chose que vous pouvez généralement ignorer.

Il peut être surprenant d'apprendre cela `Pin<&mut F>` et de l' `Pin<Box<F>>` implémenter `Unpin` , même si ce `F` n'est pas le cas. Cela ne se lit pas bien—comment est-ce `Pin` possible `Unpin` ?—mais si vous réfléchissez bien à la signification de chaque terme, cela a du sens. Même s'il `F` n'est pas sûr de se déplacer une fois qu'il a été interrogé, un pointeur vers celui-ci peut toujours être déplacé en toute sécurité, interrogé ou non. Seul le pointeur se déplace ; son référent fragile reste en place.

Ceci est utile pour savoir quand vous souhaitez passer le futur d'une fonction ou d'un bloc asynchrone à une fonction qui n'accepte que `Unpin` les futurs. (De telles fonctions sont rares dans `async_std` , mais moins ailleurs dans l'écosystème asynchrone.) `Pin<Box<F>>` est `Unpin` même si ce `F` n'est pas le cas, donc l'application `Box::pin` à une fonction asynchrone ou à un futur de bloc vous donne un futur que vous pouvez utiliser n'importe où, au prix d'une allocation de tas.

Il existe diverses méthodes non sécurisées pour travailler avec `Pin` qui vous permettent de faire ce que vous voulez avec le pointeur et sa cible, même pour les types de cible qui ne sont pas `Unpin` . Mais comme expliqué au chapitre 22 , Rust ne peut pas vérifier que ces méthodes sont utilisées correctement ; vous devenez responsable d'assurer la sécurité du code qui les utilise.

Quand le code asynchrone est-il utile ?

Asynchronele code est plus délicat à écrire que le code multithread. Vous devez utiliser les bonnes primitives d'E/S et de synchronisation, décomposer manuellement les calculs de longue durée ou les répartir sur d'autres threads, et gérer d'autres détails comme l'épinglage qui n'apparaissent pas dans le code threadé. Alors, quels avantages spécifiques le code asynchrone offre-t-il ?

Deux affirmations que vous entendrez souvent ne résistent pas à une inspection minutieuse :

- "Le code asynchrone est idéal pour les E/S." Ce n'est pas tout à fait exact. Si votre application passe son temps à attendre des E/S, la rendre asynchrone ne fera pas que ces E/S s'exécutent plus rapidement. Il n'y a rien dans les interfaces d'E/S asynchrones généralement

utilisées aujourd'hui qui les rende plus efficaces que leurs homologues synchrones. Le système d'exploitation a le même travail à faire dans les deux cas. (En fait, une opération d'E/S asynchrone qui n'est pas prête doit être réessayée plus tard, il faut donc deux appels système pour se terminer au lieu d'un.)

- "Le code asynchrone est plus facile à écrire que le code multithread." Dans des langages comme JavaScript et Python, cela pourrait bien être vrai. Dans ces langages, les programmeurs utilisent async/wait comme une forme de concurrence bien comportée : il y a un seul thread d'exécution, et les interruptions ne se produisent qu'au niveau `await` des expressions, donc il n'y a souvent pas besoin d'un mutex pour garder les données cohérentes : n'attendez pas pendant que vous êtes en train de l'utiliser ! Il est beaucoup plus facile de comprendre votre code lorsque les changements de tâche ne se produisent qu'avec votre autorisation explicite.
  Mais cet argument ne s'applique pas à Rust, où les threads ne sont pas aussi gênants. Une fois votre programme compilé, il est exempt de courses de données. Le comportement non déterministe se limite aux fonctionnalités de synchronisation telles que les mutex, les canaux, les atomes, etc., qui ont été conçues pour y faire face. Ainsi, le code asynchrone n'a pas d'avantage unique pour vous aider à voir quand d'autres threads pourraient vous affecter ; c'est clair dans *tout* code Rust sécurisé.
  Et bien sûr, le support asynchrone de Rust brille vraiment lorsqu'il est utilisé en combinaison avec des threads. Ce serait dommage d'y renoncer.

Alors, quels sont les réels avantages du code asynchrone ?

- *Les tâches asynchrones peuvent utiliser moins de mémoire.* Sous Linux, l'utilisation de la mémoire d'un thread commence à 20 Kio, en comptant à la fois l'espace utilisateur et l'espace noyau. [2] Les contrats à terme peuvent être beaucoup plus petits : les contrats à terme de notre serveur de discussion ont une taille de quelques centaines d'octets et sont devenus plus petits à mesure que le compilateur Rust s'améliore.
- *Les tâches asynchrones sont plus rapides à créer.* Sous Linux, la création d'un thread prend environ 15 µs. Générer une tâche asynchrone prend environ 300 ns, soit environ un cinquantième du temps.
- *Les changements de contexte sont plus rapides entre les tâches asynchrones qu'entre les threads du système d'exploitation* , 0,2 µs contre 1,7 µs sous Linux. [3] Cependant, il s'agit de chiffres dans le meilleur des cas pour chacun : si le commutateur est dû à la disponibilité des E/S, les deux coûts augmentent à 1,7 µs. Que le basculement se fasse entre des threads ou des tâches sur différents cœurs de processeur fait égale-

ment une grande différence : la communication entre les cœurs est très lente.

Cela nous donne un indice sur les types de problèmes que le code asynchrone peut résoudre. Par exemple, un serveur asynchrone peut utiliser moins de mémoire par tâche et être ainsi capable de gérer plus de connexions simultanées. (C'est probablement là que le code asynchrone tire sa réputation d'être "bon pour les E/S".) les changements de contexte rapides sont tous des avantages importants. C'est pourquoi les serveurs de chat sont l'exemple classique de la programmation asynchrone, mais les jeux multi-joueurs et les routeurs réseau seraient probablement aussi de bons usages.

Dans d'autres situations, les arguments en faveur de l'utilisation de l'asynchrone sont moins clairs. Si votre programme dispose d'un pool de threads effectuant des calculs lourds ou restant inactifs en attendant la fin des E/S, les avantages énumérés précédemment n'ont probablement pas une grande influence sur ses performances. Vous devrez optimiser votre calcul, trouver une connexion Internet plus rapide ou faire autre chose qui affecte réellement le facteur limitant.

En pratique, chaque récit de mise en œuvre de serveurs à volume élevé que nous avons pu trouver mettait l'accent sur l'importance de la mesure, du réglage et d'une campagne incessante pour identifier et supprimer les sources de conflit entre les tâches. Une architecture asynchrone ne vous laissera pas ignorer tout ce travail. En fait, bien qu'il existe de nombreux outils prêts à l'emploi pour évaluer le comportement des programmes multithreads, les tâches asynchrones Rust sont invisibles pour ces outils et nécessitent donc leurs propres outils.. (Comme un ancien sage a dit un jour : « Maintenant, vous avez *deux* problèmes. »)

Même si vous n'utilisez pas de code asynchrone maintenant, il est bon de savoir que l'option est là si jamais vous avez la chance d'être beaucoup plus occupé que vous ne l'êtes maintenant.

[1] Si vous avez réellement besoin d'un client HTTP, envisagez d'utiliser l'un des nombreux excellents crates comme `surf` ou `reqwest` qui fera le travail correctement et de manière asynchrone. Ce client parvient principalement à obtenir des redirections HTTPS.

[2] Cela inclut la mémoire du noyau et compte les pages physiques allouées pour le thread, et non les pages virtuelles qui n'ont pas encore été allouées. Les chiffres sont similaires sur macOS et Windows.

[3] Les commutateurs de contexte Linux étaient également dans la plage de 0,2 µs, jusqu'à ce que le noyau soit contraint d'utiliser des techniques plus lentes en raison de failles de sécurité du processeur.