Chapitre 20. Programmation asynchrone

Supposons que vous écriviez 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 de groupe de discussion à suivre, etc. Gérer tout cela pour de nombreuses connexions simultanément va prendre une certaine organisation.

Idéalement, vous pouvez simplement démarrer un thread distinct 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 fonction, qui est capable de se concentrer sur la gestion des besoins d'une seule connexion. `serve`

Cela fonctionne bien, jusqu'à ce que tout se passe beaucoup mieux que prévu et que vous ayez soudainement des dizaines de milliers d'utilisateurs. Il n'est pas rare que la pile d'un thread atteigne 100 Kio 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 demandes de mémoire sont telles que nous avons souvent besoin de moyens complémentaires, utilisés avec des threads, pour décomposer le travail.

Vous pouvez utiliser des *tâches asynchrones* Rust pour intercaler 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, passent le contrôle entre elles plus efficacement et ont une surcharge de mémoire inférieure 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 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 beaucoup moins importante que celle des threads.

Généralement, le code Rust asynchrone ressemble beaucoup au code multithread ordinaire, sauf que les opérations qui pourraient bloquer, comme les E/S ou l'acquisition de mutex, doivent être gérées un peu différemment. Le traitement de ceux-ci donne à Rust plus d'informations sur la façon dont votre code se comportera, 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 les modules de mise en réseau et de tâche de la caisse et ajoute après les appels qui peuvent être bloqués. Mais la structure globale est la même que la version basée sur les threads. `async_std` `.await`

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

- Pour montrer la mécanique de la programmation asynchrone, nous présentons un ensemble minimal de fonctionnalités de langage qui couvre tous les concepts de base: futurs, fonctions asynchrones, expressions, tâches et exécuteurs. `await` `block_on` `spawn_local`
- Ensuite, nous présentons les blocs asynchrones et l'exécuteur. Ceux-ci sont essentiels pour faire un travail réel, mais conceptuellement, ce ne

sont que des variantes des fonctionnalités que nous venons de mentionner. Dans le processus, nous soulignons quelques problèmes que vous êtes susceptible de rencontrer qui sont uniques à la programmation asynchrone et expliquons comment les gérer. `spawn`
- Pour montrer toutes ces pièces travaillant 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 et . `spawn_blocking block_on`
- Enfin, nous expliquons le type, qui apparaît de temps en temps dans les interfaces asynchrones pour s'assurer que la fonction asynchrone et les futurs de bloc sont utilisés en toute sécurité. `Pin`

Du synchrone à l'asynchrone

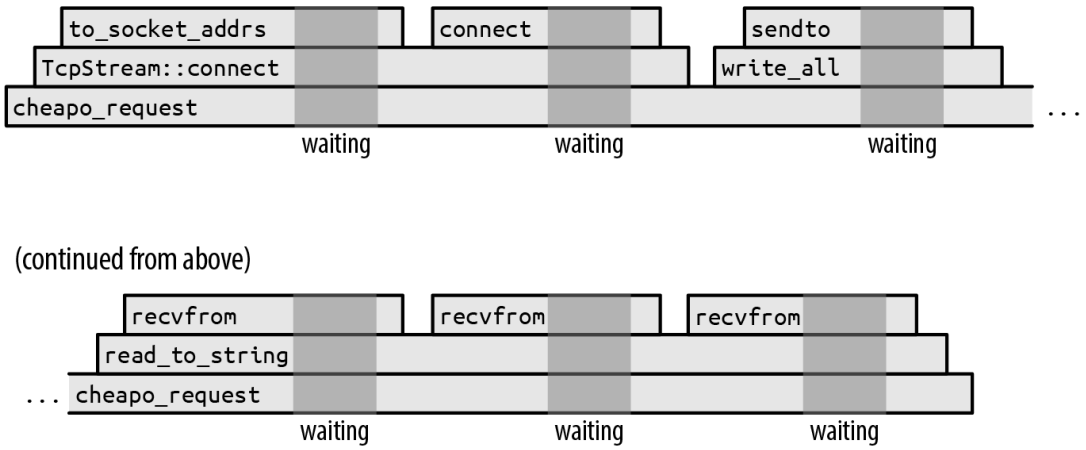
Considérez ce qui se passe lorsque vous appelez la fonction suivante (non asynchrone, complètement traditionnelle) :

```
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, h
    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 nue dans un protocole obsolète,[1] puis lit la réponse. La figure 20-1 montre l'exécution de cette fonction au fil du 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 fonction

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



Graphique 20-1. Progression d'une requête HTTP synchrone (des zones grises plus sombres attendent le système d'exploitation)

Les arrière-plans gris plus 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, l'ensemble du 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 éclats étroits entre les appels système.

Pendant que cette fonction attend le retour des appels système, son thread unique est bloqué : il ne peut rien faire d'autre tant que l'appel système n'est pas terminé. Il n'est pas rare 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 à des tâches similaires, verrouiller les ressources de ces threads pour ne rien faire d'autre que d'attendre pourrait devenir assez coûteux.

Pour contourner ce problème, un thread doit être en mesure d'effectuer d'autres tâches en attendant la fin des appels système. 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 la suivante :

```
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 aurons besoin d'une nouvelle bibliothèque d'E/S qui fournit *une version asynchrone* de cette fonction.

Contrats à terme

L'approche de Rust pour soutenir 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::Ou
}

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

A représente une opération que vous pouvez tester pour l'achèvement. La méthode d'un futur n'attend jamais la fin de l'opération : elle revient toujours immédiatement. Si l'opération est terminée, renvoie , où est son résultat final. Sinon, il renvoie . Si et quand l'avenir vaut la peine d'être interrogé à nouveau, il promet de nous le faire savoir en appelant un *waker*, une fonction de rappel fournie dans le . 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 un jusqu'à ce qu'une valeur
tombe. `Future poll poll Poll::Ready(output) output Pending C ontext poll`

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 écritures renvoient une erreur si elles se bloquent ; vous devez réessayer plus tard.

Ainsi, une version asynchrone de aurait une signature à peu près comme ceci: `read_to_string`

```
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 d'un* . Vous devrez sonder cet avenir jusqu'à ce que vous en obteniez un. Chaque fois qu'il est interrogé, la lecture se poursuit aussi loin que possible. La finale vous donne la valeur de réussite 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 a un wrapped autour d'elle. `Result<usize> Ready(result) result Future`

Appeler cette version de ne lit rien en fait; sa seule responsabilité est de construire et de rendre un avenir qui fera le vrai travail lorsqu'il sera sondé. Cet avenir doit contenir toutes les informations nécessaires à l'exécution de la demande faite par l'appel. Par exemple, l'avenir renvoyé par celui-ci doit se souvenir du flux d'entrée sur lequel il a été appelé et duquel il doit ajouter les données entrantes. En fait, puisque l'avenir contient les références et , la signature appropriée pour doit être: `read_to_string read_to_string String self buf read_to_string`

```
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 rendement futur ne peut vivre que tant que les valeurs qui empruntent et empruntent. `self buf`

La caisse fournit des versions asynchrones de toutes les installations d'E/S de , y compris un trait asynchrone avec une méthode. suit de près la conception de , réutilisant 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 familiarité avec vous aide à utiliser , et vice versa. `async-std std Read read_to_string async-std std std std std async-std`

L'une des règles du trait est que, une fois qu'un avenir est revenu, il peut supposer qu'il ne sera plus jamais interrogé. Certains contrats à terme reviennent pour toujours s'ils sont surpolés; d'autres peuvent paniquer ou pendre. (Ils ne doivent toutefois pas violer la sécurité de la mémoire ou du thread, ni provoquer un comportement non défini.) La méthode de l'adaptateur sur le trait transforme n'importe quel avenir en un avenir qui revient simplement pour toujours. Mais toutes les façons habituelles

de consommer des futurs respectent cette règle, donc n'est généralement pas
nécessaire. `Future Poll::Ready Poll::Pending fuse Future Poll::Pending fuse`

Si les sondages semblent inefficaces, ne vous inquiétez pas. L'architecture asynchrone de Rust est soigneusement conçue pour que, tant que vos fonctions d'E/S de base sont correctement implémentées, vous n'interrogerez un avenir que lorsque cela en vaudra la peine. Chaque fois qu'on l'appelle, quelque chose quelque part devrait revenir, 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? ». `read_to_string poll Ready`

Mais l'utilisation des contrats à terme semble être un défi : lorsque vous sondez, que devez-vous faire lorsque vous obtenez ? Vous devrez vous promener pour un autre travail que ce fil peut faire pour le moment, sans oublier de revenir sur ce futur plus tard et de le sonder à nouveau. L'ensemble de votre programme sera envahi par la plomberie en gardant une trace de qui est en attente et de ce qui devrait être fait une fois qu'ils sont prêts. La simplicité de notre fonction est
ruinée. `Poll::Pending cheapo_request`

Bonne nouvelle! Ce n'est pas le cas.

Fonctions asynchrones et expressions Await

Voici une version de écrit en tant que *fonction asynchrone*
: `cheapo_request`

```
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, h
    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)
}
```

C’est token pour token le même que notre version originale, sauf:

- La fonction commence par au lieu de . `async fn fn`
- Il utilise les versions asynchrones de la caisse de , et . Ceux-ci renvoient tous des futurs de leurs résultats. (Les exemples de cette section utilisent la version de .) `async_std TcpStream::connect write_all read_to_string 1.7 async_std`
- Après chaque appel qui renvoie un futur, le code indique . Bien que cela ressemble à une référence à un champ struct nommé , il s’agit en fait d’une syntaxe spéciale intégrée dans le langage pour attendre qu’un avenir soit prêt. Une expression évalue jusqu’à la valeur finale de l’avenir. C’est ainsi que la fonction obtient les résultats de , et `..await await await connect write_all 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 sera un avenir d’un , et le corps de 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 comme une fonction qui renvoie un futur d’un , pas un directement. `response std::io::Result<String> cheapo_request async fn f(...) -> T T T`

Le futur renvoyé par une fonction asynchrone enveloppe 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 pile de l’appel comme une valeur Rust ordinaire.) Il faut donc conserver les valeurs passées pour , , et , puisque le corps de ' va en avoir besoin pour fonctionner. `response host port path cheapo_request`

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 savez à ce sujet, c'est qu'il implémente , où est le type de retour de la fonction asynchrone. En ce sens, les futurs des fonctions asynchrones sont comme des fermetures : les fermetures ont également des types anonymes, générés par le compilateur, qui implémentent le , et des traits. `Future<Output=R>` `R` `FnOnce` `Fn` `FnMut`

Lorsque vous interrogez pour la première fois le futur renvoyé par , l'exécution commence en haut du corps de la fonction et s'exécute jusqu'au premier du futur renvoyé par . L'expression sonde l'avenir, et si elle n'est pas prête, alors elle retourne à son propre appelant: l'avenir du sondage ne peut pas aller au-delà de cela d'abord jusqu'à ce qu'un sondage du futur revienne. Donc, un équivalent approximatif de l'expression pourrait être: `cheapo_request` `await` `TcpStream::connect` `await` `connect P oll::Pending` `cheapo_request` `await` `TcpStream::connect` `Poll:: Ready` `TcpStream::connect(...).await`

```
{
    // 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 expression s'approprie l'avenir et l'interroge ensuite. S'il est prêt, la valeur finale du futur est la valeur de l'expression, et l'exécution se poursuit. Sinon, il renvoie le à son propre appelant. `await` `await` `Poll::Pending`

Mais surtout, le prochain sondage de l'avenir de 'ne recommence pas en haut de la fonction: au lieu de cela, il *reprend* l'exécution à mi-fonction au point où il est sur le point de sonder . Nous ne progressons pas vers le reste de la fonction asynchrone tant que cet avenir n'est pas prêt. `cheapo_request` `connect_future`

Au fur et à mesure que l'avenir de ' continue d'être sondé, il se frayera un chemin à travers le corps de la fonction de l'un à l'autre, ne se déplaçant

que lorsque le sous-futur qu'il attend sera prêt. Ainsi, le nombre de fois que l'avenir doit être interrogé dépend à la fois du comportement des sous-futurs et du flux de contrôle de la fonction. suit le point auquel le prochain devrait reprendre, et tout l'état local – variables, arguments, temporaires – dont la reprise aura
besoin. `cheapo_request await cheapo_request cheapo_request po ll`

La possibilité de suspendre l'exécution en milieu de fonction, puis de reprendre plus tard est unique aux fonctions asynchrones. Lorsqu'une fonction ordinaire revient, son cadre de pile a disparu pour de bon. Étant donné que les expressions dépendent de la possibilité de reprendre, vous ne pouvez les utiliser que dans des fonctions asynchrones. `await`

Au moment d'écrire ces lignes, Rust ne permet pas encore aux traits d'avoir des méthodes asynchrones. Seules les fonctions libres et 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 devez définir des traits qui incluent des fonctions asynchrones, envisagez d'utiliser la caisse, qui fournit une solution de contournement basée sur des macros. `async-trait`

Appel de fonctions asynchrones à partir de code synchrone : block_on

Dans un sens, les fonctions asynchrones ne font que se renvoyer la balle. Certes, il est facile d'obtenir la valeur d'un avenir dans une fonction asynchrone: juste elle. Mais la fonction asynchrone *elle-même* renvoie un avenir, c'est donc maintenant le travail de l'appelant de faire le sondage d'une manière ou d'une autre. En fin de compte, quelqu'un doit réellement attendre une valeur. `await`

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

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

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

Puisqu'il 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 devez jamais utiliser dans une fonction asynchrone: il bloquerait tout le thread jusqu'à ce que la valeur soit prête. Utilisez plutôt. `block_on block_on await`

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

La chronologie supérieure, « Vue simplifiée », affiche une vue abstraite des appels asynchrones du programme : d'abord des appels pour obtenir un socket, puis des appels et sur ce socket. Puis il revient. Ceci est très similaire à la chronologie de la version synchrone de plus haut dans ce chapitre. `cheapo_request TcpStream::connect write_all read_to _string cheapo_request`

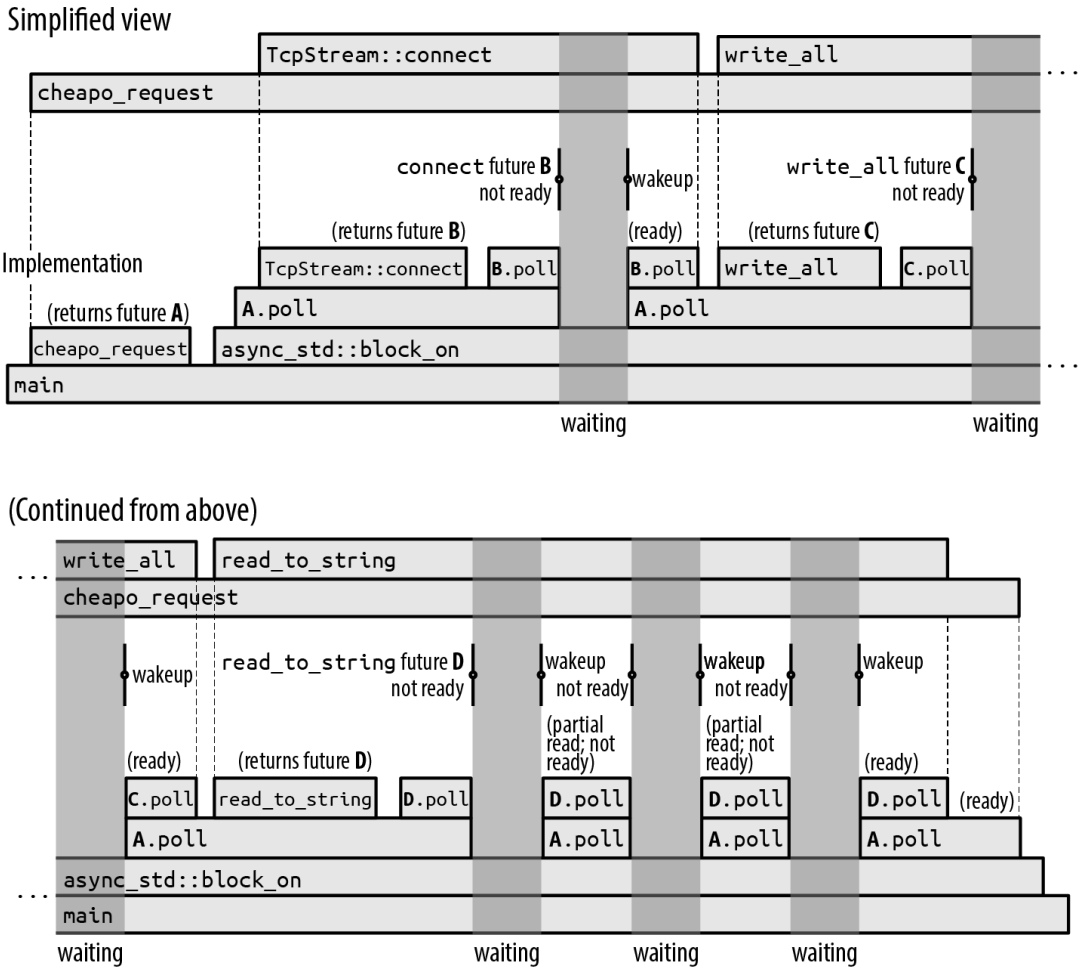


Figure 20-2. Blocage sur une fonction asynchrone

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 dans le processus. La chronologie inférieure, « Implémentation », indique les appels synchrones réels qui implémentent ce comportement asynchrone. C'est une

bonne occasion de passer en revue exactement ce qui se passe dans l'exécution asynchrone ordinaire :

- Tout d'abord, les appels , qui renvoie l'avenir de son résultat final. Puis passe cet avenir à , qui le sonde. `main cheapo_request A main async_std::block_on`
- L'interrogation future permet au corps de commencer l'exécution. Il appelle à obtenir un avenir d'une prise et l'attend ensuite. Plus précisément, puisque pourrait rencontrer une erreur, est un futur d'un `.A cheapo_request TcpStream::connect B TcpStream::connect B Result<TcpStream, std::io::Error>`
- Future est sondé par le . Étant donné que la connexion réseau n'est pas encore établie, renvoie , mais s'arrange pour réveiller la tâche d'appel une fois que le socket est prêt. `B await B.poll Poll::Pending`
- Puisque l'avenir n'était pas prêt, retourne à son propre appelant, `.B A.poll Poll::Pending block_on`
- Puisqu'il n'a rien de mieux à faire, il s'endort. L'ensemble du thread est bloqué maintenant. `block_on`
- Lorsque la connexion est prête à être utilisée, elle réveille la tâche qui l'a interrogée. Cela se met en action, et il essaie de sonder à nouveau l'avenir. `B block_on A`
- Le sondage provoque la reprise dans son premier , où il sonde à nouveau. `A cheapo_request await B`
- Cette fois, est prêt : la création du socket est terminée, il revient donc à `.B Poll::Ready(Ok(socket)) A.poll`
- L'appel asynchrone à est maintenant terminé. La valeur de l'expression est donc `.TcpStream::connect TcpStream::connect(...).await Ok(socket)`
- L'exécution du corps de 'se déroule normalement, en créant la chaîne de requête à l'aide de la macro et en la transmettant à `.cheapo_request format! socket.write_all`
- Puisqu'il s'agit d'une fonction asynchrone, elle renvoie un futur de son résultat, qui attend dûment. `socket.write_all C cheapo_request`

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

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

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

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 ont besoin d'un , mais ce ne sont que des appels de fonction. Le succès du support asynchrone de Rust dépend de l'aide aux programmeurs pour travailler avec la vue simplifiée dans la pratique, sans être distraits par les allers-retours de l'implémentation. `await`

Génération de tâches asynchrones

La fonction bloque jusqu'à ce que la valeur d'un futur 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 un autre travail* pendant qu'il attend. `async_std::task::block_on`

Pour cela, vous pouvez utiliser . Cette fonction prend un avenir et l'ajoute à un pool qui essaiera d'interroger chaque fois que l'avenir sur lequel elle bloque n'est pas prêt. Donc, si vous passez un tas de futurs à et appliquez ensuite à un futur de votre résultat final, interrogerez chaque futur engendré chaque fois qu'il sera en mesure de progresser, en exécutant l'ensemble du pool simultanément jusqu'à ce que votre résultat soit

prêt. async_std::task::spawn_local block_on spawn_local bloc
k_on block_on

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

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

La fonction est un analogue asynchrone de la fonction de la bibliothèque standard pour le démarrage des threads : spawn_local std::thread::spawn

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

Par exemple, supposons que nous voulions créer simultanément un ensemble complet 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 chaque élément de , passant l'avenir de chaque appel à . Il recueille les s résultants dans un vecteur puis attend chacun d'eux. Il est bon d'attendre les poignées de jointure dans n'importe quel ordre : puisque les requêtes sont déjà générées, leurs futurs seront interrogés au besoin chaque fois que ce thread appelle et n'a rien de mieux à faire. Toutes les demandes s'exécuteront simultanément. Une fois qu'ils sont terminés, renvoie les résultats à son appelant. `cheapo_request requests spawn_local JoinHandle block_on many_requests`

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

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

    handles.push(task::spawn_local(cheapo_request(&host, port, &path)));
                                   ---------------^^^^^--------------
                                   |              |
                                   |              borrowed value does no
                                   |              live long enough
                       argument requires that `host` is borrowed for `'sta
}
- `host` dropped here while still borrowed
```

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

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

Le problème est que vous ne pouvez pas être sûr que vous attendrez que la tâche se termine avant et que vous êtes abandonné. En fait, n'accepte que les futurs dont la durée de vie est , car vous pouvez simplement ignorer les retours 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 pour démarrer un thread dont la fermeture capture des références à des variables locales. `spawn_local host path spawn_local 'static JoinHandle std::thread::spawn`

Une façon de résoudre ce problème consiste à créer une autre fonction asynchrone qui prend les 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 s au lieu de références, de sorte que son futur possède le et les cordes lui-même, et sa durée de vie est . Le vérificateur d'emprunt peut voir qu'il attend immédiatement l'avenir de 's, et par conséquent, si cet avenir est sondé du tout, les variables qu'il emprunte doivent toujours exister. Tout va bien. `String &str host path 'static cheapo_request host path`

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

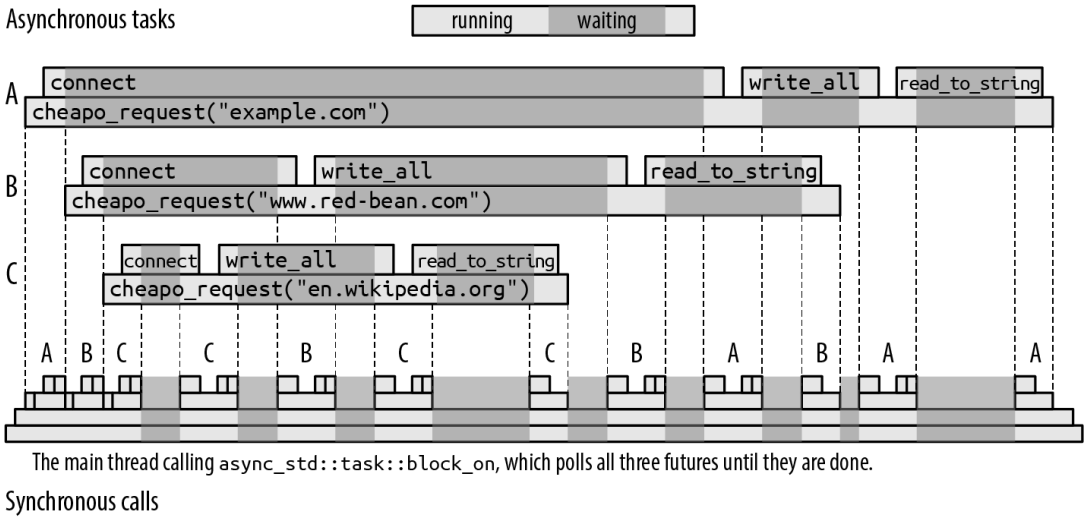
```
for (host, port, path) in requests {
    handles.push(task::spawn_local(cheapo_owning_request(host, port, pat
}
```

Vous pouvez appeler depuis votre fonction synchrone, avec : `many_requests main 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 demandes simultanément à partir de l'appel à . Chacun progresse au fur et à mesure que l'occasion se présente tandis que les autres sont bloqués, le tout sur le fil d'appel. La figure 20-3 montre une exécution possible des trois appels à . `block_on cheapo_request`

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



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

L'appel à (non affiché, pour plus de simplicité) a engendré trois tâches asynchrones, que nous avons étiquetées , et . commence par l'interrogation , qui commence à se connecter à . Dès que cela revient, tourne son attention vers la tâche suivante engendrée, l'interrogation future , et finalement , qui commencent chacun à se connecter à leurs serveurs respectifs. `many_requests A B C block_on A example.com Poll::Pending block_on B C`

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

Dans cette exécution, le serveur 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 la sienne et la marque comme prête, afin qu'elle puisse continuer quand elle l'attend. Finalement, les autres appels à réussiront ou retourneront une erreur, et lui-même peut revenir. Enfin, reçoit le vecteur de résultats de `.en.wikipedia.org JoinHandle many_requests cheapo_request many_requests main block_on`

Toute cette exécution se déroule sur un seul fil, les trois appels à être entrelacés les uns avec les autres à travers des sondages successifs de leur avenir. Un appel asynchrone offre l'apparence d'un appel de fonction unique s'exécutant jusqu'à la fin, mais cet appel asynchrone est réalisé par une série d'appels synchrones à la méthode du futur. Chaque appel

individuel revient rapidement, ce qui donne le fil de discussion afin qu'un autre appel asynchrone puisse prendre un tour. `cheapo_request poll poll`

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

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 des expressions, lorsque l'avenir attendu revient . Cela signifie que si vous mettez un calcul de longue durée dans , aucune des autres tâches que vous avez passées n'aura la chance de s'exécuter tant qu'elle n'est pas terminée. Avec les threads, ce problème ne se pose pas : le système d'exploitation peut suspendre n'importe quel thread à n'importe quel point et définit des minuteries pour s'assurer qu'aucun thread ne monopolise le processeur. Le code asynchrone dépend de la coopération volontaire des futurs partageant le thread. Si vous avez besoin que des calculs de longue durée coexistent avec du code asynchrone, « Long Running Computations: yield_now and spawn_blocking » plus loin dans ce chapitre décrit certaines options. `await Poll::Pending cheapo_request spawn_local`

Blocs asynchrones

En plus des 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 expressions dans un bloc asynchrone. `await`

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

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

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

Si un bloc asynchrone fait référence à des variables définies dans le code environnant, son avenir capture leurs valeurs, tout comme le ferait une fermeture. Et tout comme les fermetures (voir « Fermetures qui volent »), vous pouvez commencer le bloc avec pour prendre possession des valeurs capturées, plutôt que de simplement conserver des références à celles-ci. `move` `async move`

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, nécessitait un futur, nous avons donc défini la fonction wrapper pour nous donner un futur qui s'appropriait ses arguments. Vous pouvez obtenir le même effet sans la distraction d'une fonction wrapper simplement en appelant à partir d'un bloc asynchrone : `spawn_local` `'static` `cheapo_owning_request` `cheapo_request`

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

Un inconvénient 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 arguments suivants d'une fonction asynchrone. Cela peut entraîner des problèmes lors de l'utilisation de l'opérateur : `-> T ?`

```
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 méthode renvoie , mais comme l'opérateur utilise le trait pour convertir le type d'erreur en question en fonction de la situation requise, le type de retour du bloc asynchrone peut être pour n'importe quel type qui im-

plémente `. read_line Result<(), std::io::Error> ? From Result<(), E> E From<std::io::Error>`

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

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

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

Création de fonctions asynchrones à partir de blocs asynchrones

Les blocs asynchrones 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 exemple comme une fonction synchrone ordinaire qui renvoie l'avenir d'un bloc asynchrone : `cheapo_request`

```
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 l'avenir de la valeur du bloc asynchrone. Cela capture les arguments de la fonction et se comporte comme le futur que la fonction asynchrone aurait renvoyé. Comme nous n'utilisons pas la syntaxe, nous devons écrire le 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. `async fn impl Future`

Cette deuxième approche peut être utile lorsque vous souhaitez effectuer un calcul immédiatement lorsque la fonction est appelée, avant de créer l'avenir de son résultat. Par exemple, une autre façon de se réconcilier avec serait d'en faire une fonction synchrone renvoyant un futur qui capture des copies entièrement détenues de ses arguments
: `cheapo_request 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 et en tant que valeurs possédées, et non en tant que références. Étant donné que le futur possède toutes les données dont il a besoin pour s'exécuter, il est valable pour la durée de vie. (Nous l'avons précisé dans la signature montrée précédemment, mais c'est la valeur par défaut pour les types de retour, donc l'omettre n'aurait aucun
effet.) `host path String &str 'static + 'static 'static -> impl`

Puisque cette version des rendements futures qui sont , on peut les passer directement à : `cheapo_request 'static 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 exemples que nous avons montrés jusqu'à présent passent presque tout leur temps à attendre les E/S, mais certaines charges de travail sont plutôt un mélange de travail de processeur et de blocage. Lorsque vous

avez suffisamment de calcul pour faire ce qu'un seul processeur ne peut pas suivre, vous pouvez l'utiliser pour générer un avenir sur un pool de threads de travail dédiés à l'interrogation des futurs qui sont prêts à progresser. `async_std::task::spawn`

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

En pratique, est plus largement utilisé que , simplement parce que les gens aiment savoir que leur charge de travail, quel que soit son mélange de calcul et de blocage, est équilibrée dans les ressources de la machine. `spawn spawn_local`

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

Si vous utilisez le stockage local de thread, il peut être surprenant de voir les données que vous y placez avant une expression remplacées par quelque chose de complètement différent par la suite, car votre tâche est maintenant interrogée par un thread différent du pool. S'il s'agit d'un

problème, vous devez plutôt utiliser *le stockage local de tâche* ; voir la documentation de la caisse pour la macro pour plus de détails. `await async-std task_local!`

Mais votre future implémentation envoie-t-elle ?

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

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

Cette restriction est facile à contourner 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());
```

L'avenir d'une fonction asynchrone doit contenir suffisamment d'informations pour que la fonction puisse continuer à partir d'une expression. Dans ce cas, le futur de ' doit utiliser après le , de sorte que le futur contiendra, au moins parfois, une valeur. Étant donné que les pointeurs ne peuvent pas être partagés en toute sécurité entre les threads, l'avenir lui-même ne peut pas être . Et puisque n'accepte que les futurs qui sont , les

objets Rust

: await reluctant string await Rc<String> Rc Send spawn Send

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

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

```
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 place de . utilise des mises à jour atomiques pour gérer ses nombres de références, ce qui le rend un peu plus lent, mais les pointeurs sont
. `std::sync::Arc Rc Arc Arc Send`

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

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

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

```
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, en pointant vers le type comme coupable. Puisque Rust considère que le résultat de est présent pour l'ensemble de l'instruction, y compris l'expression, il détermine que l'avenir de n'est pas . Cette erreur est trop prudente de la part de Rust: bien qu'il soit vrai qu'il n'est pas sûr d'envoyer à un autre thread, le seul se produit lorsque le résultat est , de sorte que la valeur d'erreur n'existe jamais réellement lorsque nous attendons l'avenir de
`. Result some_fallible_thing match await unfortunate Send Ge nericError await Ok use_output`

La solution idéale consiste à utiliser des types d'erreur 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 le type d'erreur sous-jacent pour être implémenté , et tout va bien. `Send`

Si votre avenir ne l'est pas et que vous ne pouvez pas le faire facilement, vous pouvez toujours l'utiliser pour l'exécuter sur le fil de discussion actuel. Bien sûr, vous devrez vous assurer que le thread appelle à un moment donné, pour lui donner une chance de s'exécuter, et vous ne bénéficierez pas de la distribution du travail sur plusieurs processeurs. `Send spawn_local block_on`

Calculs de longue durée : yield_now et spawn_blocking

Pour qu'un futur partage bien son fil avec d'autres tâches, sa méthode doit toujours revenir le plus rapidement possible. Mais si vous effectuez un long calcul, cela peut prendre beaucoup de temps pour atteindre le suivant, ce qui fait que d'autres tâches asynchrones attendent plus longtemps que vous ne le souhaiteriez pour leur activation du thread. `poll await`

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

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

La première fois que l'avenir est sondé, il revient, mais dit que cela vaut la peine de sonder 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 deuxième fois, l'avenir de 's est interrogé, il retourne et votre fonction asynchrone peut reprendre
l'exécution. `yield_now Poll::Pending yield_now Poll::Ready(())`

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

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

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 être intensive en calcul afin que, même si les attaquants obtiennent une copie de notre base de données, ils ne puissent pas simplement essayer des milliards de mots de passe possibles pour voir s'ils correspondent. La caisse fournit une fonction de hachage conçue spécifiquement pour stocker les mots de passe: un hachage correctement généré prend une fraction de seconde importante à vérifier. Nous pouvons utiliser (version) dans notre application asynchrone comme ceci: `argonautica argonautica argonautica 0.2`

```
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 si correspond , étant donné , une clé pour la base de données dans son ensemble. En effectuant la vérification dans la fermeture passée à , 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. `Ok(true)` `password` `hash` `key` `spawn_blocking`

Comparaison des conceptions asynchrones

À bien des égards, l'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 expressions. Et tous ces langages ont des valeurs qui représentent des calculs incomplets : Rust les appelle « futures », JavaScript les appelle « promesses » et C# les appelle « tâches », mais ils représentent tous une valeur que vous devrez peut-être attendre. `await`

L'utilisation des sondages par Rust, cependant, est inhabituelle. En JavaScript et C#, une fonction asynchrone commence à s'exécuter dès qu'elle est appelée, et une boucle d'événements globale est 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 transmettiez à une fonction comme , , ou qui l'interroge et conduise le travail à son terme. Ces fonctions, *appelées exécuteurs*, jouent le rôle que d'autres langages couvrent avec une boucle d'événements globale. `block_on` `spawn` `spawn_local`

Parce que Rust vous oblige, en tant que programmeur, à choisir un exécuteur pour sonder vos avenirs, Rust n'a pas besoin d'une boucle d'événements globale intégrée au système. La caisse offre les fonctions d'exécuteur que nous avons utilisées dans ce chapitre jusqu'à présent, mais la caisse, que nous utiliserons plus loin dans ce chapitre, définit son propre ensemble de fonctions d'exécuteur similaires. Et vers la fin de ce chapitre, nous mettrons en œuvre notre propre exécuteur testamentaire. Vous pouvez utiliser les trois dans le même programme. `async-std tokio`

Un véritable client HTTP asynchrone

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

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

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

Ensuite, nous pouvons définir comme suit: `many_requests`

```
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),
        }
    }
}
```

L'utilisation d'un seul pour effectuer toutes nos requêtes nous permet de réutiliser les connexions HTTP si plusieurs d'entre elles sont dirigées vers le même serveur. Et aucun bloc asynchrone n'est nécessaire : puisqu'il s'agit d'une méthode asynchrone qui renvoie un futur, on peut passer son futur directement à . `surf::Client recv_string Send + 'static spawn`

Un client et un serveur asynchrones

Il est temps de prendre les idées clés dont nous avons discuté jusqu'à présent et de les rassembler dans un programme de travail. Dans une large mesure, les applications asynchrones ressemblent à des applications multithread ordinaires, mais il existe de nouvelles possibilités de code compact et expressif que vous pouvez rechercher.

L'exemple de cette section est un serveur et un client de chat. Consultez 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 les nôtres 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 comme un client lent ne doit pas obliger le serveur à dépenser de la mémoire illimitée en conservant son arriéré de messages toujours croissant, notre serveur doit supprimer les messages pour les clients qui ne peuvent pas suivre, mais les informer 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.)

Nous commençons le projet avec la commande et mettons le texte suivant dans *async-chat/Cargo.toml*: `cargo new --lib async-chat`

```
[package]
name = "async-chat"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2021"

[dependencies]
async-std = { version = "1.7", features = ["unstable"] }
tokio = { version = "1.0", features = ["sync"] }
serde = { version = "1.0", features = ["derive", "rc"] }
serde_json = "1.0"
```

Nous dépendons de quatre caisses :

- La caisse est la collection de primitives d'E/S asynchrones et d'utilitaires que nous avons utilisés tout au long du chapitre. `async-std`
- La caisse est une autre collection de primitives asynchrones comme , l'une des plus anciennes et des plus matures. Il est largement utilisé et maintient sa conception et sa mise en œuvre à des normes élevées, mais nécessite un peu plus de soin à utiliser que . `tokio async-std async-std`
  `tokio` est une grande caisse, mais nous n'en avons besoin que d'un seul composant, de sorte que le champ de la ligne de dépendance *Cargo.toml* se réduit aux pièces dont nous avons besoin, ce qui en fait une dépendance légère. `features = ["sync"] tokio`
  Lorsque l'écosystème des bibliothèques asynchrones était moins mature, les gens évitaient d'utiliser les deux et 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. `tokio async-std`
- Les caisses que nous avons déjà vues, 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 , nous les sélectionnons donc lorsque nous donnons la dépendance. `serde serde_json serde`

Toute la structure 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 mise en page de paquet utilise une fonctionnalité Cargo que nous avons abordée dans « Le répertoire src / bin »: en plus de la caisse de bibliothèque principale, *src / lib.rs*, avec son sous-module *src / utils.rs*, il comprend également deux exécutables:

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

Nous présenterons le contenu de chaque fichier source au cours du chapitre, mais une fois qu'ils sont tous en place, si vous tapez dans cette arborescence, cela compile la caisse de bibliothèque, 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 des définitions partagées par le client et le serveur. De même, vérifie l'ensemble de l'arborescence source. Pour exécuter l'un des exécutables, vous pouvez utiliser des commandes telles que celles-ci : `cargo build cargo check`

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

L'option indique l'exécutable à 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 du serveur et le port TCP. `-- bin --`

Types d'erreurs et de résultats

Le module de la caisse de bibliothèque définit les types de résultats et d'erreurs que nous utiliserons dans toute l'application. *Depuis src/utils.rs* : `utils`

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

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

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

Dans une application réelle, envisagez d'utiliser la caisse, qui fournit et des types similaires à ceux-ci. La caisse est facile à utiliser et offre de belles fonctionnalités au-delà de ce que nous pouvons offrir. `anyhow Error Result anyhow ChatError ChatResult`

Le Protocole

La caisse de bibliothèque capture l'ensemble de notre protocole de chat dans 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'énumération représente les paquets qu'un client peut envoyer au serveur : il peut demander à rejoindre un groupe et publier des messages dans n'importe quel groupe qu'il a rejoint. représente ce que le serveur peut renvoyer : les messages publiés dans un groupe et les messages d'erreur. L'utilisation d'une référence comptée au lieu d'une simple aide le serveur à éviter de faire des copies de chaînes lorsqu'il gère des groupes et distribue des
messages. `FromClient FromServer Arc<String> String`

Les attributs indiquent à la caisse de générer des implémentations de ses et traits 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 formulaires Rust. `# [derive] serde Serialize Deserialize FromClient FromServer s erde_json::to_string serde_json::from_str`

Le test unitaire illustre comment cela est utilisé. Compte tenu de l'implémentation dérivée par , nous pouvons appeler pour transformer la valeur donnée en ce

JSON: `test_fromclient_json Serialize serde serde_json::to_string FromClient`

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

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

Prise en compte des entrées utilisateur : flux asynchrones

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

La méthode asynchrone est intéressante. Il ne peut pas renvoyer un itérateur, comme le fait la bibliothèque standard : la méthode est une fonction synchrone ordinaire, donc l'appel bloquerait le thread jusqu'à ce que la ligne suivante soit prête. Au lieu de cela, renvoie un *flux* de 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 trait, à partir du module: `BufReader lines Iterator::next command_lines.next() l ines Result<String> Stream async_std::stream`

```
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 de la et des traits. Comme un itérateur, a a un type associé et utilise pour indiquer quand la séquence est terminée. Mais comme un futur, un flux doit être interrogé : pour obtenir l'élément suivant (ou apprendre que le flux est terminé), vous devez appeler jusqu'à ce qu'il revienne. L'implémentation d'un flux doit toujours revenir rapidement, sans blocage. Et si un flux revient, il doit avertir l'appelant lorsqu'il vaut la peine d'interroger à nouveau via le

fichier
`. Iterator Future Stream Item Option poll_next Poll::Ready po ll_next Poll::Pending Context`

La 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 et . Parmi ceux-ci se trouve une méthode, qui renvoie un avenir du prochain flux. Plutôt que d'interroger explicitement le flux, vous pouvez appeler et attendre l'avenir qu'il renvoie à la
place. `poll_next filter map next Option<Self::Item> next`

En assemblant ces éléments, on consomme le flux de lignes d'entrée en bouclant les valeurs produites par un flux en utilisant avec
: `send_commands next while let`

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

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

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

Lorsque vous travaillez avec des flux, n'oubliez pas d'utiliser le prélude
: `async_std`

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

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

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

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

Ceci est un exemple du modèle de *trait d'extension* que nous avons décrit dans « Traits et types d'autres personnes ». Le module met les méthodes en champ d'application, de sorte que l'utilisation du prélude garantit que ses méthodes sont visibles dans votre code. `async_std::prelude StreamExt`

Envoi de paquets

Pour transmettre des paquets sur une prise réseau, notre client et notre serveur utilisent la fonction du module de notre caisse de bibliothèque : `send_as_json utils`

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

pub async fn send_as_json<S, P>(outbound: &mut S, packet: &P) -> ChatRes
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 génère la représentation JSON de comme un , ajoute une nouvelle ligne à la fin, puis écrit le tout dans . `packet String outbound`

D'après son article, vous pouvez voir que c'est assez souple. Le type de paquet à envoyer, , peut être tout ce qui implémente . Le flux de sortie peut être tout ce qui implémente , la version asynchrone du trait pour les flux de sortie. Ceci est suffisant pour que nous envoyions et des valeurs sur un fichier asynchrone . Garder la définition de générique garantit qu'elle ne dépend pas des détails du flux ou des types de paquets de manière surprenante: ne peut utiliser que des méthodes à partir de ces traits. `where send_as_json P serde::Serialize S async_std::io:`

`:Write std::io::Write FromClient FromServer TcpStream send_as_json send_as_json`

La contrainte sur est requise pour utiliser la méthode. Nous aborderons l'épinglage et le dépouillement plus loin dans ce chapitre, mais pour le moment, il devrait suffire d'ajouter des contraintes pour taper des variables si nécessaire; le compilateur Rust signalera ces cas si vous oubliez. `Unpin S write_all Unpin`

Plutôt que de sérialiser le paquet directement dans le flux, sérialisez-le dans un temporaire, puis l'écrit dans . La caisse fournit des fonctions permettant de 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 modifications fondamentales à la fois et au cœur indépendant du format de la caisse, car les caractéristiques autour desquelles elles sont conçues ont des méthodes synchrones. `outbound send_as_json String outbound serde_json serde_json serde_json serde`

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

Réception de paquets : plus de flux asynchrones

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

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

pub fn receive_as_json<S, P>(inbound: S) -> impl Stream<Item = ChatResul
    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 , cette fonction est générique dans les types de flux d'entrée et de paquets : `send_as_json`

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

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

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

Cela indique que nous *retournons un* type qui produit une séquence de 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 a de toute façon un type anonyme, c'est le type le plus spécifique qui pourrait éventuellement revenir. `ChatResult<P> map receive_as_json`

Notez qu'il ne s'agit pas, en soi, d'une fonction asynchrone. C'est une fonction ordinaire qui renvoie une valeur asynchrone, un flux. Comprendre la mécanique du support asynchrone de Rust plus profondément que « simplement ajouter et partout » ouvre la possibilité de définitions claires, flexibles et efficaces comme celle-ci qui tirent pleinement parti du langage. `receive_as_json async .await`

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

```
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 un socket recevant des données du serveur, en enroule un autour de celui-ci (notez bien, la version), puis les transmet pour obtenir un flux de valeurs entrantes. Ensuite, il utilise une boucle pour gérer les réponses entrantes, vérifier les résultats des erreurs et imprimer chaque réponse du serveur pour que l'utilisateur puisse la voir. `BufReader async_std receive_as_json FromServer while let`

Fonction principale du client

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

```
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, dispose d'une série de fonctions asynchrones qu'il aimerait appeler, de sorte qu'il enveloppe le reste de la fonction dans un bloc asynchrone et passe le futur du bloc à exécuter. `main` `async_std::task::block_on`

Une fois la connexion établie, nous voulons que les fonctions et 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 devrait se fermer. `send_commands` `handle_replies`

Compte tenu de ce que nous avons fait ailleurs dans le chapitre, vous pouvez 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 comme nous attendons les deux poignées de jointure, cela nous donne un programme qui se ferme une fois *les deux* tâches terminées. Nous voulons sortir dès que *l'un ou l'autre* aura terminé. La méthode sur les futurs accomplit cela. L'appel renvoie un nouvel avenir qui sonde à la fois et revient 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 terminé en premier. L'avenir inachevé est abandonné. `race` `from_server.race(to_server)` `from_server` `to_server` `Poll::Ready(v)`

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

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

Fonction principale du serveur

Voici tout le contenu du fichier principal pour le 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 fonction du serveur ressemble à celle du client: il fait un peu de configuration, puis appelle pour exécuter un bloc asynchrone qui fait le vrai travail. Pour gérer les connexions entrantes des clients, il crée un socket, dont la méthode renvoie un flux de valeurs. `main block_on TcpListener incoming std::io::Result<TcpStream>`

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

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

Gestion des connexions de chat : Mutex asynchrones

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

```
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_n
                        }
                    }
                }
            };

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

        Ok(())
    }
```

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

En plus des messages d'erreur, les clients souhaitent 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 tout le monde un clone de , mais si deux de

ces sources essaient 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. `TcpStream`

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

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

Lorsqu'elle est créée, une valeur prend possession d'un et l'enveloppe dans un pour s'assurer qu'une seule tâche peut l'utiliser à la fois. La fonction encapsule chacun dans un pointeur compté par référence afin que tous les groupes joints par le client puissent pointer vers la même instance
partagée. `Outbound TcpStream Mutex serve Outbound Arc Outboun d`

Un appel à verrouiller d'abord le mutex, renvoyant une valeur de garde qui déréférence à l'intérieur. Nous avons l'habitude de transmettre , puis finalement nous appelons pour nous assurer qu'il ne languira pas à moitié transmis dans un tampon quelque part. (À notre connaissance, ne tamponne pas réellement les données, mais le trait permet à ses implémentations de le faire, nous ne devrions donc prendre aucun risque.) `Outbound::send TcpStream send_as_json packet guard.f lush() TcpStream Write`

L'expression nous permet de contourner le fait que Rust n'applique pas de coercitions deref pour répondre aux limites des traits. Au lieu de cela, nous déréférençons explicitement la garde mutex, puis empruntons une

référence mutable à la protection qu'elle protège, produisant ce qui nécessite. `&mut *guard TcpStream &mut TcpStream send_as_json`

Notez que cela utilise le type, pas le fichier . Il y a trois raisons à cela. `Outbound async_std::sync::Mutex Mutex`

Tout d'abord, la bibliothèque standard peut mal se comporter si une tâche est suspendue tout en tenant une garde mutex. Si le thread qui avait exécuté cette tâche récupère une autre tâche qui tente de verrouiller la même chose, des problèmes s'ensuivent: du point de vue du point de vue, le thread qui le possède déjà essaie de le verrouiller à nouveau. La norme n'est pas conçue pour gérer ce cas, il panique donc ou 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 garde est en direct sur une expression. Puisqu'il doit tenir la serrure pendant qu'elle attend les futurs de et , il doit utiliser 's
. `Mutex Mutex Mutex Mutex std::sync::Mutex await Outbound::send send_as_json guard.flush async_std Mutex`

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

Enfin, la norme 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'implémente pas : il ne peut pas être transmis à d'autres threads. Cela signifie qu'un futur tenant une telle garde ne s'implémente pas lui-même , et ne peut pas être passé à s'exécuter sur un pool de threads; il ne peut être exécuté qu'avec ou . La garde pour un implémente donc il n'y a aucun problème à l'utiliser dans les tâches
engendrées. `Mutex Send Send spawn block_on spawn_local async_std Mutex Send`

La table de groupe : Mutex synchrones

Mais la morale de l'histoire n'est pas aussi simple que « Toujours utiliser dans du code asynchrone ». Souvent, il n'est pas nécessaire d'attendre

quoi que ce soit tout en tenant un mutex, et la serrure n'est pas maintenue longtemps. Dans de tels cas, la bibliothèque standard peut être beaucoup plus efficace. Le type de notre serveur de chat illustre ce cas. Voici le contenu complet de *src/bin/server/group_table.rs*: `async_std::sync::Mutex Mutex GroupTable`

```
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 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 et verrouillent le mutex, effectuent quelques opérations de table de hachage, peut-être quelques allocations, et retournent. `GroupTable get get_or_create`

Dans , nous utilisons un vieux simple . Il n'y a pas de code asynchrone dans ce module, il n'y a donc pas de s à éviter. En effet, si nous voulions utiliser ici, nous aurions besoin de faire et dans des fonctions asynchrones, ce qui introduit la surcharge de la création future, des suspensions et des reprises pour peu d'avantages: le mutex n'est verrouillé que

pour certaines opérations de hachage et peut-être quelques allocations. `GroupTable` `std::sync::Mutex` `await` `async_std::sync::Mutex` `get` `get_or_create`

Si notre serveur de chat se retrouvait avec des millions d'utilisateurs et que le 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 . Par exemple, la caisse fournit un tel type. `GroupTable` `HashMap` `dashmap`

Groupes de discussion : les chaînes de diffusion de tokio

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

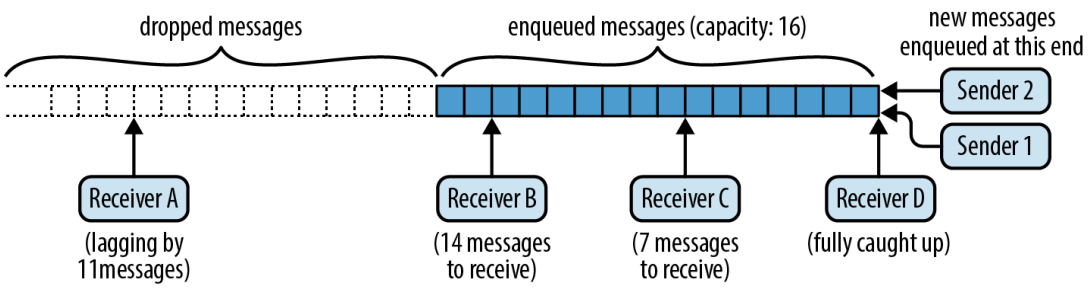
C'est là que nous abordons le défi mentionné plus tôt de la *contre-pression*. Il y a plusieurs besoins en tension les uns avec les autres:

- Si un membre ne peut pas suivre les messages publiés dans le groupe (s'il a une connexion réseau lente, par exemple), les autres membres du groupe ne doivent pas être affectés.
- Même si un membre prend du retard, il devrait y avoir des moyens pour lui de se joindre à la conversation et de continuer à participer d'une manière ou d'une autre.
- La mémoire 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 caisse fournit un type de *canal de diffusion* qui implémente un ensemble raisonnable de compromis. Un canal de diffusion est une file d'attente de valeurs (dans notre cas, des messages de chat) qui permet à un nombre illimité de fils de discussion ou de tâches différents d'envoyer et de recevoir des valeurs. C'est ce qu'on appelle un canal de « diffusion » car chaque consommateur obtient sa propre copie de chaque valeur envoyée. (Le type de valeur doit être implémenté.) `tokio` `tokio` `Clone`

Normalement, un canal de diffusion conserve 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 n'ont pas pu suivre reçoivent une erreur la prochaine fois qu'ils essaient d'obtenir leur prochain message, et le canal les rattrape au message le plus ancien encore disponible.

Par exemple, la figure 20-4 montre un canal de diffusion d'une capacité maximale de 16 valeurs.



Graphique 20-4. Une chaîne de diffusion tokio

Il y a deux expéditeurs qui mettent les messages en file d'attente et quatre destinataires qui les mettent en file d'attente, ou plus précisément, qui 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 complètement rattrapé. Le récepteur A a pris du retard et 11 messages ont été supprimés 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 valeurs : la publication d'un message au groupe le diffuse à tous les membres actuels. Voici la définition du type, définie dans *src/bin/server/group.rs* : `Arc<String> group::Group`

```
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 subscriptio
            // slightly before its incoming side, which may end up trying to
            // message to an empty group.
            let _ignored = self.sender.send(message);
        }
    }
```

Une structure contient le nom du groupe de discussion, ainsi qu'une `diffusion::Expéditeur` représentant l'extrémité d'envoi du canal de diffusion du groupe. La fonction appelle à créer un canal de diffusion d'une capacité maximale de 1 000 messages. La fonction renvoie à la fois un expéditeur et un récepteur, mais nous n'avons pas besoin du récepteur à ce stade, car le groupe n'a pas encore de membres. `Group Group::new broadcast::channel channel`

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

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

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 : il s'agit d'une boucle qui reçoit des messages du canal de diffusion et les transmet au client via la valeur partagée. Si la boucle ne peut pas suivre le canal de diffusion, elle reçoit une erreur, qu'elle signale consciencieusement au client. `Outbound Lagged`

Si l'envoi d'un paquet au client échoue complètement, peut-être parce que la connexion est fermée, quitte sa boucle et revient, ce qui entraîne la fermeture de la tâche asynchrone. Cela supprime le canal de diffusion, en le désabonnant de la chaîne. De cette façon, lorsqu'une connexion est interrompue, chacune de ses appartenances à un groupe est nettoyée la prochaine fois que le groupe tente de lui envoyer un message. `handle_subscriber Receiver`

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

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

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

Futurs primitifs et exécuteurs testamentaires: quand un avenir vaut-il la peine d'être sondé à nouveau?

Le serveur de chat montre comment nous pouvons écrire du code en utilisant des primitives asynchrones comme et le canal, et utiliser des exécuteurs comme et pour piloter leur exécution. Maintenant, nous pouvons jeter un coup d'œil à la façon dont ces choses sont mises en œuvre. La question clé est, lorsqu'un futur revient, comment se coordonne-t-il avec l'exécuteur testamentaire pour l'interroger à nouveau au bon moment? `TcpListener broadcast block_on spawn Poll::Pending`

Pensez à ce qui se passe lorsque nous exécutons du 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?;
    ...
})
```

La première fois qu'elle sonde l'avenir du bloc asynchrone, la connexion réseau n'est presque certainement pas prête immédiatement, alors elle se met en veille. Mais quand devrait-il se réveiller? D'une manière ou d'une autre, une fois que la connexion réseau est prête, doit dire qu'il doit essayer d'interroger à nouveau l'avenir du bloc asynchrone, car il sait que cette fois, la volonté se terminera et l'exécution du bloc asynchrone peut progresser. `block_on block_on TcpStream block_on await`

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

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

```
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 de l'avenir est prête, retournez-la. Sinon, cachez un clone du waker de 's quelque part, et revenez `. Context Poll::Pending`

Lorsque l'avenir vaut à nouveau la peine d'être interrogé, l'avenir doit en informer le dernier exécuteur testamentaire qui l'a interrogé en appelant la méthode de son waker : `wake`

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

Idéalement, l'exécuteur testamentaire et le futur se relaient pour sonder et se réveiller : l'exécuteur sonde l'avenir et s'endort, puis l'avenir invoque le réveil, de sorte que l'exécuteur se réveille et sonde à nouveau l'avenir.

Les futurs des fonctions et des blocs asynchrones ne traitent pas des réveils 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 réveils. Dans notre client de chat, le premier sondage de l'avenir du bloc asynchrone ne fait que transmettre le contexte lorsqu'il attend l'avenir de . Les sondages suivants transmettent également leur contexte à l'avenir que le bloc attend ensuite. `TcpStream::connect`

`TcpStream::connect` ' l'avenir gère l'interrogation comme le montre l'exemple précédent : il remet le réveil à un thread d'assistance qui attend que la connexion soit prête, puis appelle le réveil.

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

Il est inoffensif pour un exécuteur testamentaire de surplomber un avenir, tout simplement inefficace. Les futurs, cependant, devraient faire attention à n'invoquer un réveil que lorsque l'interrogation ferait des progrès réels: un cycle de réveils et de sondages fallacieux peut empêcher un exécuteur de dormir du tout, gaspillant de l'énergie et laissant le processeur moins réactif à d'autres tâches.

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

Invoquer les wakers : spawn_blocking

Plus haut dans le chapitre, nous avons décrit la fonction, qui démarre une fermeture donnée en cours d'exécution sur un autre thread et renvoie un futur de sa valeur de retour. Nous avons maintenant toutes les pièces dont nous avons besoin pour 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 le fait la version de
. `spawn_blocking` `spawn_blocking` `async_std`

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

La signature de notre est la suivante: `spawn_blocking`

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

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

`SpawnBlocking<T>` est un avenir de la valeur de rendement 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 structure doit servir de rendez-vous entre le futur et le fil qui exécute la fermeture, elle est donc détenue par un et protégé par un . (Un mutex synchrone est bien ici.) L'interrogation de l'avenir vérifie si elle est présente et enregistre le réveil dans le cas contraire. Le thread qui exécute la fermeture enregistre sa valeur de retour dans, puis appelle , le cas échéant. `Shared` `Arc` `Mutex` `value` `waker` `value` `waker`

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 valeur, cela génère un thread pour exécuter la fermeture, stocker le résultat dans le champ de ' et appeler le waker, le cas échéant. `Shared Shared value`

Nous pouvons mettre en œuvre pour ce qui suit: `Future SpawnBlocking`

```
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 est vérifie si la valeur de la fermeture est prête, en prenant possession et en la renvoyant si c'est le cas. Sinon, l'avenir est toujours en attente, il enregistre donc un clone du waker du contexte dans le champ du futur. `SpawnBlocking waker`

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

mise en œuvre block_on

En plus de 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 . Ce sera un peu plus simple que la version de '; par exemple, il ne prend pas en charge , les variables locales de tâche ou les appels imbriqués (appel à partir de code asynchrone). Mais il suffit d'exécuter notre client et serveur de chat. `block_on async_std spawn_local block_on`

Voici le code :

```
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.

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

Le type de caisse est une primitive de blocage simple : l’appel bloque le thread jusqu’à ce que quelqu’un d’autre appelle le correspondant, que vous obtenez au préalable en appelant . Si vous avez un thread qui n’est pas encore garé, son prochain appel à revenir immédiatement, sans blocage. Notre volonté d’attendre chaque fois que l’avenir n’est pas prêt, et le réveil que nous passons aux futurs le dégarera. `crossbeam Parker parker.park() .unpark() Unparker parker.unparker() unpark park block_on Parker`

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

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

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

Étant donné qu’une variable détient un futur de type , la macro prend possession du futur et déclare une nouvelle variable du même nom dont le type est et qui emprunte le futur. Cela nous donne l’exigence de la méthode. Pour les raisons que nous expliquerons dans la section suivante, les futurs des fonctions et blocs asynchrones doivent être référencés via un avant de pouvoir être interrogés. `F pin! Pin<&mut F> Pin<&mut Self> poll Pin`

```
    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 sillage, nous sondons l'avenir jusqu'à ce qu'il revienne. S'il retourne , nous garons le thread, qui bloque jusqu'à ce qu'il soit appelé. Ensuite, nous réessayons. `Poll::Ready Poll::Pending waker`

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

Épinglage

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

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

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 , tout ce que le serveur veut envoyer. Les points étiquetés ❶, ❷ et ❸ sont les *points de reprise*, les points du code de la fonction asynchrone auxquels l'exécution peut être suspendue. `String`

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

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

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

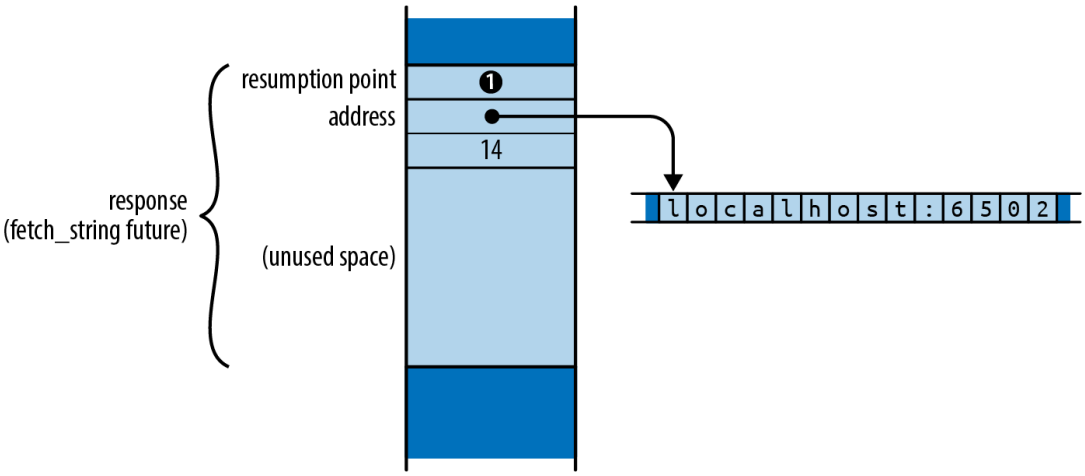


Figure 20-5. L'avenir construit pour un appel à `fetch_string`

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

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

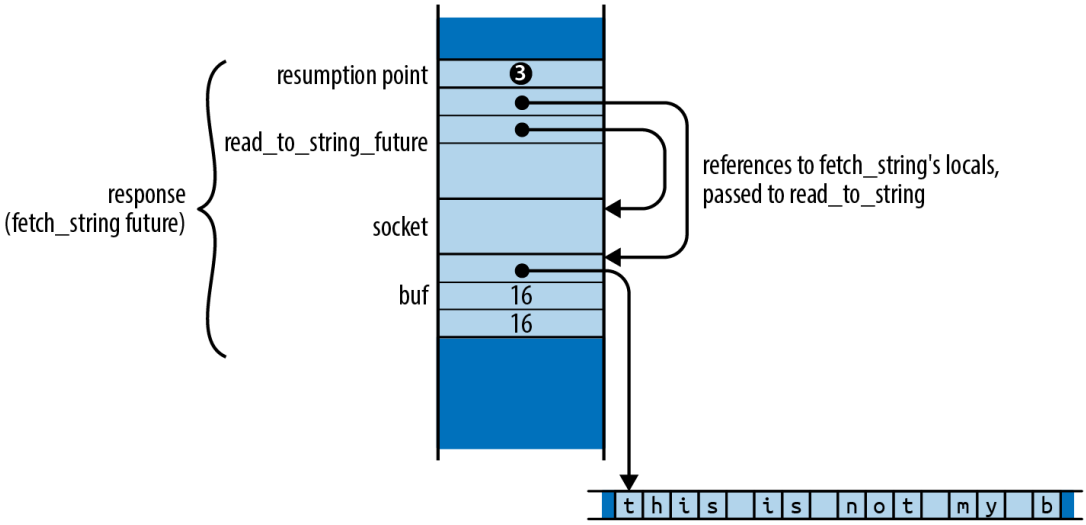
```
socket.read_to_string(&mut buf).await❸?;
```

Supposons en outre que le résultat de n'est pas prêt, de sorte que le sondage retourne . À ce stade, l'avenir ressemble à la figure 20-6. `read_to_string` `Poll::Pending`

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 l'avenir du scrutin. `await read_to_string`
- Les variables qui sont vivantes à ce point de reprise : et . La valeur de n'est plus présente dans le futur, puisque la fonction n'en a plus besoin. `socket` `buf` `address`

- Le sous-futur, dont l'expression est au milieu des sondages. `read_to_string await`



Graphique 20-6. Le même avenir, en pleine attente `read_to_string`

Notez que l'appel à a emprunté des références à et . Dans une fonction synchrone, toutes les variables locales vivent sur la pile, mais dans une fonction asynchrone, les variables locales qui sont vivantes sur un doivent être localisées à l'avenir, de sorte qu'elles seront disponibles lorsqu'elles seront à nouveau interrogées. Emprunter une référence à une telle variable emprunte une partie de l'avenir. `read_to_string socket buf await`

Cependant, Rust exige que les valeurs ne soient pas déplacées pendant qu'elles sont empruntées. Supposons que vous deviez déplacer cet avenir 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 et vers leurs nouveaux emplacements, les références continuent de pointer vers leurs anciens emplacements dans le maintenant non initialisé. Ils sont devenus des pointeurs pendants, comme le montre la figure 20-7. `socket buf response`

Empêcher le déplacement des valeurs empruntées relève généralement de 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 et affectent non seulement ce qui peut faire avec ses propres variables, mais ce que son appelant peut faire en toute

sécurité avec , l'avenir qui les réserve. Les futurs des fonctions asynchrones sont 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. `socket buf fetch_string response`

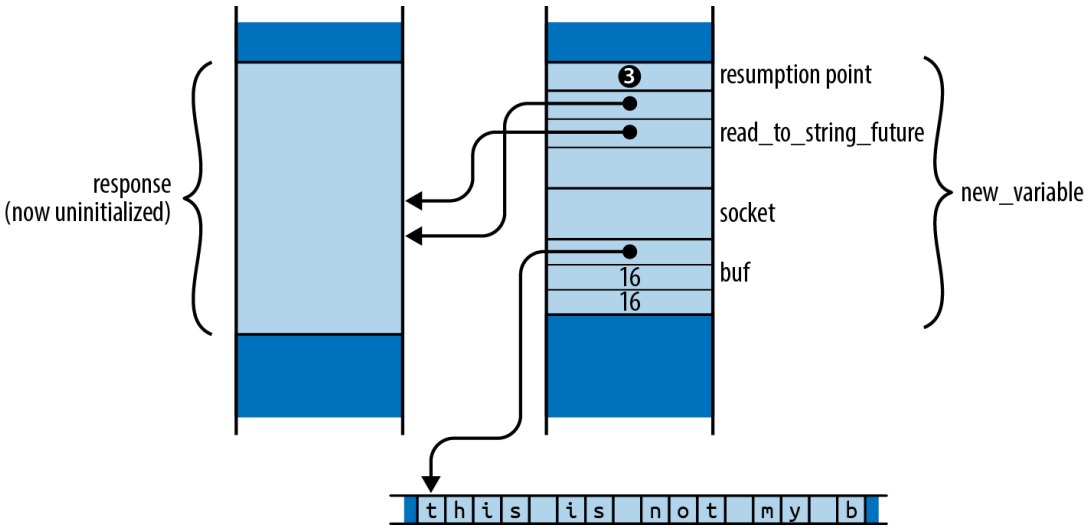


Figure 20-7. (') , déplacé alors qu'il est emprunté (Rust empêche cela) `fetch_string`

La solution de Rust à ce problème repose sur l'idée que les futurs sont toujours sûrs à se 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 d'argument. Ceux-ci ne sont que dans la portée du corps de la fonction asynchrone, qui n'a pas encore commencé l'exécution. Seul le sondage d'un avenir peut emprunter son contenu.

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

- La première étape commence lorsque le futur est créé. Parce que le corps de la fonction n'a pas encore commencé à s'exécuter, aucune partie de celle-ci ne pourrait encore être empruntée. À ce stade, il est aussi sûr de se déplacer que n'importe quelle autre valeur rust.
- La deuxième étape commence la première fois que l'avenir est sondé. Une fois que le corps de la fonction a commencé à s'exécuter, il pourrait 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 pourrait ne pas être sûr de bouger.

La flexibilité de la première étape de la vie est ce qui nous permet de passer des futures à et et appeler des méthodes d'adaptateur comme et , qui prennent toutes des futures par valeur. En fait, même l'appel de fonc-

tion asynchrone qui a créé l'avenir en premier lieu devait le renvoyer à l'appelant; c'était aussi une mesure. `block_on spawn race fuse`

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

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

« Une valeur que vous ne pouvez pas déplacer » semble impossible: les mouvements sont partout dans Rust. Nous expliquerons exactement comment protéger les futurs dans la section suivante. `Pin`

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

Gardez à l'esprit que cette fragilité de déplacement est limitée aux futurs de fonctions et de blocs asynchrones, avec leurs implémentations spéciales générées par le compilateur. Si vous implémentez à la main pour vos propres types, comme nous l'avons fait pour notre type dans « Invoking Wakers: spawn_blocking », ces futurs sont parfaitement sûrs à déplacer avant et après avoir été interrogés. Dans toute implémentation manuscrite, le vérificateur d'emprunt s'assure que toutes les références que vous aviez empruntées à des parties de sont parties disparues au moment du 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. `Future Future SpawnBlocking poll self poll`

Pointeurs épinglés

Le type est un wrapper pour les pointeurs vers les futurs qui limite la façon dont les pointeurs peuvent être utilisés pour s'assurer que les futurs 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. `Pin`

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

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

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

Notez que le champ *n'est pas* . Cela signifie que la seule façon de construire ou d'utiliser un est à travers les méthodes soigneusement choisies que le type fournit. `pointer pub Pin`

Compte tenu de l'avenir d'une fonction ou d'un bloc asynchrone, il n'y a que quelques façons d'obtenir un pointeur épinglé vers celui-ci :

- La macro, à partir de la caisse, ombre une variable de type avec une nouvelle de type . La nouvelle variable pointe vers la valeur de l'original, 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 dans notre mise en œuvre pour épingler l'avenir que nous voulions sonder. `pin! futures-lite T Pin<&mut T> pin! block_on`
- Le constructeur de la bibliothèque standard prend possession d'une valeur de n'importe quel type, la déplace dans le tas et renvoie un fichier . `Box::pin T Pin<Box<T>>`
- `Pin<Box<T>>` implémente , prend donc possession et vous redonne une boîte épinglée pointant vers la même sur le tas. `From<Box<T>> Pin::from(boxed) boxed T`

Chaque façon d'obtenir un pointeur épinglé vers ces futurs implique de renoncer à la propriété de l'avenir, 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 déplacer un pointeur ne déplace pas son référent. Ainsi, la possession d'un pointeur épinglé vers un avenir sert de preuve que

vous avez définitivement renoncé à la capacité de déplacer cet avenir. C'est tout ce que nous devons savoir pour qu'il puisse être interrogé en toute sécurité.

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

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

```
pin!(future);

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

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

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

La raison pour laquelle nous n'avons pas à nous soucier d'épingler des futurs dans du code asynchrone ordinaire est que les moyens les plus courants d'obtenir la valeur d'un futur – en l'attendant ou en passant à un exécuteur testamentaire – prennent tous possession du futur et gèrent l'épinglage en interne. Par exemple, notre implémentation s'approprie l'avenir et utilise la macro pour produire le nécessaire pour interroger. Une expression s'approprie également l'avenir et utilise une approche similaire à la macro en interne. `block_on pin! Pin<&mut F> await pin!`

Le trait Unpin

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

Ces types durables mettent en œuvre le trait de marqueur: `Unpin`

```
trait Unpin { }
```

Presque tous les types dans Rust implémentent automatiquement , en utilisant un support spécial dans le compilateur. La fonction asynchrone et les contrats à terme de bloc sont les exceptions à cette règle. `Unpin`

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

Par exemple, implémente , afin que nous puissions écrire: `String Unpin`

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

Cela signifie que lorsque vous implémentez pour vos propres types, votre implémentation peut traiter comme si elle était , pas . L'épinglage devient

quelque chose que vous pouvez la plupart du temps ignorer. `Future Unpin poll self &mut Self Pin<&mut Self>`

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

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

Il existe différentes méthodes dangereuses pour travailler avec qui vous permettent de faire ce que vous voulez avec le pointeur et sa cible, même pour les types de cible qui ne le sont pas . 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. `Pin Unpin`

Quand le code asynchrone est-il utile ?

Le code asynchrone est plus difficile à écrire que le code multithread. Vous devez utiliser les bonnes primitives d'E/S et de synchronisation, diviser les calculs de longue durée à la main ou les faire tourner sur d'autres threads, et gérer d'autres détails comme l'épinglage qui ne se produisent pas dans le code threadé. Alors, quels sont les avantages spécifiques du code asynchrone ?

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, le fait de l'asynchroniser ne permettra pas d'exécuter ces E/S 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 syn-

chrones. Le système d'exploitation a le même travail à faire dans les deux sens. (En fait, une opération d'E/S asynchrone qui n'est pas prête doit être réessayée ultérieurement, il faut donc deux appels système pour 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/await comme une forme de concurrence bien conduite : il n'y a qu'un seul thread d'exécution, et les interruptions ne se produisent qu'au niveau des expressions, il n'y a donc 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 tâches ne se produisent qu'avec votre autorisation explicite. `await`
  Mais cet argument ne se répercute pas sur Rust, où les fils 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. Le code asynchrone n'a donc aucun avantage unique pour vous aider à voir quand d'autres threads pourraient vous affecter ; c'est clair dans *tout* code Rust sûr.
  Et bien sûr, le support asynchrone de Rust brille vraiment lorsqu'il est utilisé en combinaison avec des fils. Il serait dommage d'y renoncer.

Alors, quels sont les vrais 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 futurs peuvent être beaucoup plus petits: les futurs de notre serveur de chat ont une taille de quelques centaines d'octets et sont de plus en 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. La génération d'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, ce sont les meilleurs chiffres pour chacun : si le commutateur est dû à la disponibilité des E/S, les deux coûts augmentent à 1,7 µs. Que le basculement soit entre les threads ou les tâches sur différents cœurs de processeur fait également 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 ainsi être en mesure de gérer plus de connexions simultanées. (C'est probablement là que le code asynchrone a la réputation d'être « bon pour les E/S ».) Ou, si votre conception est naturellement organisée comme de nombreuses tâches indépendantes communiquant entre elles, alors de faibles coûts par tâche, des temps de création courts et des changements de contexte rapides sont tous des avantages importants. C'est pourquoi les serveurs de chat sont l'exemple classique de programmation asynchrone, mais les jeux multi-joueurs et les routeurs réseau seraient probablement également de bonnes utilisations.

Dans d'autres situations, les arguments en faveur de l'utilisation de l'async 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.

Dans la pratique, chaque compte rendu de la mise en œuvre de serveurs à volume élevé que nous avons pu trouver soulignait 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 permettra pas d'ignorer ce travail. En fait, bien qu'il existe de nombreux outils prêts à l'emploi pour évaluer le comportement des programmes multithread, les tâches asynchrones Rust sont invisibles pour ces outils et nécessitent donc leur propre outillage. (Comme un sage aîné l'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'une des nombreuses excellentes caisses comme ou qui feront le travail correctement et de manière asynchrone. Ce client parvient principalement à obtenir des redirections HTTPS. `surf` `reqwest`

2 Cela inclut la mémoire du noyau et compte les pages physiques allouées au thread, et non les pages virtuelles qui doivent encore être allouées. Les chiffres sont similaires sur macOS et Windows.

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